CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Technical Report

Sufficient Temporal Independence and Improved Interrupt Latencies in a Real-Time Hypervisor

Matthias Beckert

Moritz Neukirchner

Rolf Ernst

Stefan M. Petters

CISTER-TR-140303

Version:

Date: 3/10/2014

Sufficient Temporal Independence and Improved Interrupt Latencies in a Real-Time Hypervisor

Matthias Beckert, Moritz Neukirchner, Rolf Ernst, Stefan M. Petters

CISTER Research Unit

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: , , , smp@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

Virtualization techniques for hard real-time systems typically employ TDMA scheduling to achieve temporal isolation among partitions. The processing of user-level interrupt handlers is only performed within appropriate time slots, thus signi_cantly increasing interrupt latencies. We propose a novel approach permitting execution of user-level interrupt handlers during time slots of other partitions hence reducing interrupt latencies. Su_cient temporal independence among partitions, as required by safety standards, is maintained through a monitoring mechanism, which bounds the interference of user-level interrupt handlers in other partitions. We show correctness of the approach and evaluate its performance in a hypervisor implementation.

Sufficient Temporal Independence and Improved Interrupt Latencies in a Real-Time Hypervisor

Matthias Beckert, Moritz Neukirchner and Rolf Ernst
Institute of Computer and Network Engineering
TU Braunschweig, Germany
{beckert | neukirchner | ernst}@ida.ing.tu-bs.de

Stefan M. Petters
CISTER/INSEC-TEC
Polytechnic Institute of Porto, Portugal
smp@isep.ipp.pt

ABSTRACT

Virtualization techniques for hard real-time systems typically employ TDMA scheduling to achieve temporal isolation among partitions. The processing of user-level interrupt handlers is only performed within appropriate time slots, thus significantly increasing interrupt latencies.

We propose a novel approach permitting execution of user-level interrupt handlers during time slots of other partitions hence reducing interrupt latencies. Sufficient temporal independence among partitions, as required by safety standards, is maintained through a monitoring mechanism, which bounds the interference of user-level interrupt handlers in other partitions. We show correctness of the approach and evaluate its performance in a hypervisor implementation.

1. INTRODUCTION

Virtualization techniques, which are well established in general purpose computing, have come of age also in the domain of embedded real-time systems. Implementations, such as PikeOS [13] or OK:Microvisor [4], are commercially available and used in relevant industries. With the Integrated Modular Avionics (IMA) architecture of the ARINC653 standard [10] virtualization techniques have become part of a standardized software architecture in safety-critical systems.

Opposed to general purpose computing, virtualized real-time systems not only require spatial isolation among partitions (typically achieved through use of virtual memory) but also sufficient temporal independence. Particularly, in safety-critical systems this is required by applicable standards such as IEC61508 [5].

Current virtualization environments achieve a complete temporal isolation through use of time-division multiple access (TDMA) scheduling [13]. The single partitions that shall be isolated from each other are assigned time slots of fixed length in which they may execute. The virtualization environment cycles through these time slots according to a fixed schedule. The cycle length, which is calculated as sum of all time slot lengths, is referred to as *TDMA cycle length*. As a result from this scheduling, the processing time each partition receives is completly independent of the behavior of other partitions.

Interrupt handling is split into two parts. The direct hardware interrupt request (IRQ) is processed by the virtualization environment in the *top handler*, which may preempt the application level processes. This includes, e.g. clearing the IRQ flags. Further, the actual application-level processing of the interrupt is performed in a *bottom handler*, which is executed in the context of a partition. Therefore, worst-case interrupt latencies, i.e. the time between occurrence of the interrupt and the end of processing in the bottom handler, are governed by the TDMA cycle length because a partition's time slot may have just ended when the interrupt occurred.

Reduction of the TDMA cycle length to reduce interrupt latencies is not always an option as this requires frequent partition switches, which may significantly increase overhead.

In this paper we propose a novel approach to reduce interrupt latencies. We relax the isolation requirements and allow to execute bottom IRQ handler within time slots of other partitions. However, in order to maintain sufficient temporal independence, as required by safety standards, we bound and enforce the maximum interference such interrupts have on other partitions. This is achieved through appropriate monitoring and shaping mechanisms that delay bottom handler processing to the depending partition in case a defined maximum interference to the system is reached. We show the correctness of the approach and evaluate it with our own modified implementation of the MicroC/OS hypervisor (uC/OS-MMU) [1].

2. RELATED WORK

Interrupt latency in virtualization environments has so far received limited attention in the literature. While in general-purpose systems the concept of interrupt latency for guest operating systems has been examined, the notion of interrupt latency in real-time embedded systems has only be investigated with the notion of reducing the interrupt latencies within the hypervisor.

For example, Ongaro et al. [9] investigated the Xen credit-based scheduler and its shortcoming for interrupt latencies. In order to address this issue, they added a new scheduling priority that is higher than that of regular schedulable partitions (domains). Whenever an interrupt is delivered via an event to any of the partitions, the receiving partition

DAC '14 June 1-5 2014, San Fransisco, CA, USA

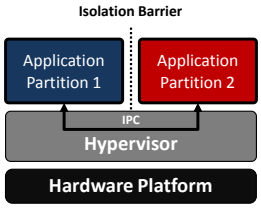


Figure 1: Hypervisor system architecture

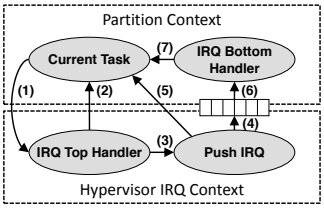


Figure 2: Interrupt Handling

is immediately placed in this higher priority class for one time slice to respond to the interrupt. This approach was refined by Kim et al. [6] to account for finer granularities than the tick based accounting of the original credit based system. Besides other works addressing similar concerns, Whiteaker et al. [19] have compared the interrupt latency observed by Xen, KVM, and a standard Linux. While the fundamental idea of boosting the performance of interrupt serving domains achieves the desired effect, the lack of temporal partition enforcement within Xen is not suitable for real-time workloads. In particular I/O devices are concentrated in these models into a separate partition, which does not allow for the desired development of isolation of guest partitions.

The work of Blackham et al. [2] focuses on shortening the interrupt latencies through manipulation of kernel operations in the formally verified seL4 kernel. However, it does not address the issue of guest operating system level interrupt/event latencies due to the introduction of partitions addressed here.

Throttling overloading interrupts at their source has been targeted by Regehr and Duongsaa [11]. This monitors incoming interrupt requests and if a pre-specified limit has been reached does not clear the interrupt flag until a new interrupt is permissible again. While it also addresses the avoidance of overloads we present in this work, it does not cover the latency concerns for guest operating systems in virtualized systems.

3. HYPERVISOR ARCHITECTURE AND INTERRUPT HANDLING

In this section we describe the underlying architecture of the hypervisor and the associated IRQ handling. The description follows the implementation of uC/OS-MMU [1], which we have used for the implementation. However, the general architecture and IRQ handling mechanisms are comparable to other hypervisors, as e.g. L4 based systems [14].

The aim of a hypervisor is to isolate different applications spatially and temporally. The general architecture of a hypervisor system is depicted in Figure 1. Applications are executed in separate *application partitions*. The hypervisor controls scheduling of the partitions, communication among partitions, and the access rights to memory and peripherals. In the scope of this paper we solely regard temporal properties.

Within each application partition a guest operating system can be executed. The proposed solution supports both full- as well as para-virtualization techniques. Without loss of generality we will focus in the discussion on the terminology for para-virtualization, in which the guest operating systems is modified to run on the virtual machine. To achieve temporal isolation between the application partitions, the hypervisor uses TDMA scheduling for the partitions, i.e. each partition p_i is assigned a time slot of fixed size T_i. The hypervisor cycles through the time slots in a static order. Unused processing capacity of a time partition is left unused even when other time partitions have outstanding jobs. This way the temporal properties (e.g. worst-case response times) of any software running within a partition does not depend on the execution behavior of other partitions.

Direct access to the interrupt controller is only possible from the context of the hypervisor in order to enforce isolation. The IRQ handling within an application partition is therefore not based on the processors interrupt controller but rather on "emulated" IRQ from the hypervisor. The concept is illustrated in Figure 2.

The basic processing of hardware IRQs (e.g. resetting IRQ flags) is directly handled by the IRQ *top handler* within the hypervisor IRQ context (1). After the IRQ is processed by the top handler, the hypervisor checks if any partition has to react on this IRQ (2,3). Further, the hypervisor pushes an event in the respective *interrupt queue* of each partition that has to react on the IRQ (4). Newer processor architectures also support a virtual interface for the interrupt controller, which is maintained by the hypervisor [18]. After pushing the IRQ to the respective queues, the hypervisor switches back to the partition context. Upon each partition context switch (i.e. everytime a partition is given the right to execute) the partition checks its interrupt queue for any pending IRQs. In case pending IRQs exist, instead of continuing from the last interruption point within the partition's context (5), the partition calls an IRQ *bottom handler* (6), which processes all pending interrupts. Only after this regular processing is resumed (7).

From a timing perspective this behavior can cause long interrupt latencies as we illustrate in Figure 3. The system is executing in partition 1 when a hardware interrupt (HW IRQ) occurs. Immediately the execution of partition 1 is interrupted and the top handler of the hypervisor is called. The top handler pushes an event in the interrupt queue of partition 2 which shall process the IRQ. Then the hypervisor returns to the context of partition 1. After the timeslot of partition 1, the system switches to partition 2 that immediately calls the bottom handler to process the pending IRQ. From this setup we see that IRQ latencies are largely determined through the assignment of the TDMA cycle and timeslot lengths. In the worst-case an IRQ occurs right after the end of the partition with the appropriate bottom handler. The aim of the approach in this paper is to reduce IRQ latencies (independent of the TDMA cycle length) while maintaining sufficient temporal independence instead of complete temporal isolation.

4. IRQ LATENCIES AND SUFFICIENT TEMPORAL INDEPENDENCE

In order to reason about IRQ latencies and sufficient temporal independence, we first formalize these concepts in this section. We start by a defining the difference between tem-

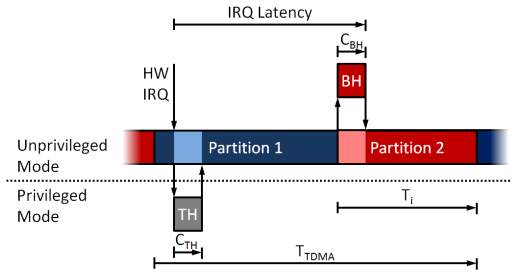


Figure 3: Interrupt Latency

poral isolation and sufficient temporal independence. Later we will provide an analysis for the worst-case IRQ latency for IRQ handling according to the above TDMA scheme and analyze the interference that IRQs impose on partitions. We profit from that a partition is nothing else than a tasks to the hypervisor's scheduler, so that standard analysis methods can be used.

For P partitions, we define a taskset $\mathrm{T} = \{\tau_1, \ldots, \tau_P\}$. To each τ_p where $p \in [1, P]$ we assign a set of tasks $\mathcal{I}_p$ that might interferer with τ_p and the overall interference I_p which is caused by $\mathcal{I}_p$. For temporal isolation no interference is allowed, which leads us to:

$$\mathcal{I}_p = \{\varnothing\} \Rightarrow I_p = 0 \tag{1}$$

In case of sufficient temporal independence, a bounded interference $\widehat{I}_p$ caused by $\mathcal{I}_p$ is allowed, where I_{p,τ_j} indicates the worst-case interference a task τ_p can suffer from a task τ_j.

$$\mathcal{I}_p = \{\mathrm{T} \setminus \tau_p\} \Rightarrow I_p = \sum_{\tau_j \in \mathcal{I}_p} I_{p,\tau_j} \leq \widehat{I}_p \tag{2}$$

In order to determine the worst-case IRQ latency, we use the concept of the *busy-window analysis* [7, 16], which is known from response-time analysis. To analyze a given task τ_i, the analysis requires the set of tasks $\mathcal{I}_i$ that can interfere with τ_i, the worst-case execution times C_i of τ_i and its interferers, and a description of their activation patterns. Activation patterns are modelled via *arrival functions* $\eta^+(\Delta t)$ [3], which return the maximum number of events that can arrive within any time window of size Δt. At some points in this paper the dual representation of *minimum distance functions* $\delta^-(q)$ [12], which yield the minimum distance of any q events, is used.

The busy-window analysis determines an upper bound on the amount of time a resource requires to service q activations of task τ_i [16]. This upper bound, the *q-event busy time* $W_i(q)$, is given through the following iterative formula, which is evaluated until convergence to a fixed-point

$$W_i(q) = q \cdot C_i + \sum_{j \in \mathcal{I}_i} C_j \cdot \eta^+(W_i(q)) \tag{3}$$

To determine the worst-case response time R_i of τ_i, the first Q_i activations of τ_i have to be considered, where

$$Q_i = \max\left(n : \forall q \in \mathbb{N}^+, q \leq n : \delta_i^-(q) \leq W_i(q-1)\right) \tag{4}$$

The worst-case response time [15] is then given as

$$R_i = \max_{q \in [1, Q_i]} \left(W_i(q) - \delta_i^-(q)\right) \tag{5}$$

Now, we provide an analysis for the worst-case interrupt latencies of the above TDMA-based interrupt handling based on the busy-window analysis. Consider the analysis of a given IRQ source IRQ_i. Each invocation of IRQ_i requires the processing of one top handler and one bottom handler. Thus, the worst-case computation time is given through

$$C_i = C_{TH_i} + C_{BH_i} \tag{6}$$

Any invocation of IRQ_i can suffer delay from three sources: 1) interference from TDMA time partitions in which IRQ_i may not execute its bottom handler (denoted by I_{TDMA}), 2) interference from top handlers of other IRQ sources (denoted by I_{TH_j}), and 3) interference from top handlers of the same IRQ source I_{TH_i}. There is no interference from bottom handlers of the same IRQ source as bottom handler invocations are processed in FIFO manner due to the queue mechanism (cf. Figure 2). Thus, for the analysis of IRQ latencies we can rewrite the q-event busy-window of IRQ_i (3), which denotes the longest time to process q activations of IRQ_i, as

$$W_i(q) = q \cdot C_i + I_{TDMA}(W_i(q)) + I_{TH_j}(W_i(q)) + I_{TH_i}(W_i(q)) \tag{7}$$

Next, we determine the individual interference terms. Let T_i be the length of the time slot in which IRQ_i may execute its bottom handler. The worst-case interference suffered through other partitions (including context switch overhead) in a time-window of Δt is [17]

$$I_{TDMA}(\Delta t) = \lceil \Delta t / T_{TDMA} \rceil \cdot (T_{TDMA} - T_i) \tag{8}$$

The worst-case interference from top handlers of other IRQ sources in Δt is determined through [16]

$$I_{TH_j}(\Delta t) = \sum_{j \in \mathcal{I}_i} \eta_j^+(\Delta t) \cdot C_{TH_j} \tag{9}$$

where η_j^+ denotes the upper arrival curve describing the arrival pattern of an interfering IRQ IRQ_j.

Now, let us consider the interference through the top handlers of the same IRQ source. For any time interval Δt the maximum number of activations of IRQ_i is given through $\eta_i^+(\Delta t)$. In a q-event busy-window q, top handler activations are already accounted for in the term $q \cdot C_i$. Therefore, the additional interference through own top handlers is given through

$$I_{TH_i}(\Delta t) = (\eta_i^+(\Delta t) - q) \cdot C_{TH_i} \tag{10}$$

Combining (6)-(10) yields

$$\begin{aligned} W_i(q) = & q \cdot C_{BH_i} + \eta_i^+(W_i(q)) \cdot C_{TH_i} \\ & + \lceil W_i(q) / T_{TDMA} \rceil \cdot (T_{TDMA} - T_i) \\ & + \sum_{j \in \mathcal{I}_i} \eta_j^+(W_i(q)) \cdot C_{TH_j} \end{aligned} \tag{11}$$

The worst-case IRQ latency is then given through

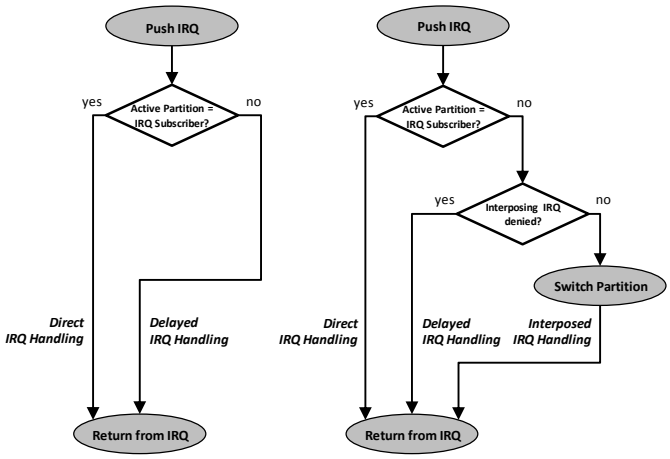
$$R_i = \max_{q \in [1, Q_i]} \left(W_i(q) - \delta_i^-(q)\right) \tag{12}$$

Typically, the TDMA cycle and timeslot lengths are large compared to C_{TH} and C_{BH} to reduce the overhead of frequent partition context switches. Typically we have

$$C_{TH}, C_{BH} \ll T_{TDMA} - T_i$$

Thus, the worst-case interrupt latency is dominated by the TDMA cycle length.

Further, we see that the worst-case latency depends on the processing of top handlers of IRQs that shall be processed in other partitions. Although this weakens temporal isolation, this is typically tolerated for several reasons. 1) Top



(a) Original top handler (b) Modified top handler

Figure 4: Top Handler

handlers have minimal execution time (i.e. resetting IRQ flags and pushing an event to the respective queue), 2) top handlers are executed in hypervisor context, and therefore the code is trusted and 3) disabling the IRQ source while not in the appropriate partition may cause missing IRQs as in most cases IRQ flags are not counting. Further, the handling of IRQs that are shared by several partitions would be particularly complicated.

In this paper we aim to reduce IRQ latencies while maintaining sufficient temporal independence of partitions, i.e. to allow processing of top handlers of other partitions but to effectively control the processing of bottom handlers.

5. MONITORING BASED IRQ SHAPING

In this section we introduce our approach to effectively reduce interrupt latencies. We do this by not only permitting top handlers but also bottom handlers to execute during other partitions' timeslots. However, execution of bottom handlers is monitored and shaped according to a predefined monitoring condition. We will show that this results in a bounded interference on other partitions and leads to greatly reduced average interrupt latencies. Also worst-case interrupt latencies are significantly reduced if interrupts do not violate the monitoring condition.

In order to monitor and shape bottom handler processing we modify the interrupt top handler of the operating system. Figure 4a shows the original top handler behavior in uC/OS-MMU. The top handler merely pushes an IRQ event to the partition's interrupt queue and returns into the original partition context. There is no distinction whether or not the interrupt is for the currently running partition. In case the partition of the interrupt is currently running (*direct IRQ handling*), the interrupt is processed immediately after returning into the unprivileged mode. Otherwise it is later executed in the correct TDMA slot (*delayed IRQ handling* as shown in Figure 3).

We have modified the top handler as shown in Figure 4b. Again the top handler pushes an IRQ event to the partition's interrupt queue. If the active partition is not the IRQ subscriber, a monitoring function (*Interposing IRQ denied?*) checks whether the requested bottom handler is allowed to execute within a foreign TDMA slot. If this is not the case, processing remains as for the unmodified top handler. Otherwise, i.e. if the activation of the requested bottom handler satisfies the monitoring condition, the top handler calls the scheduler, switches into the bottom handler's partition for C_{BH_i} time units and returns into the unprivileged mode, i.e. the bottom handler is called and may execute for at most C_{BH_i} time units. This maximum execution time is enforced through the hypervisor by calling the partition scheduler after C_{BH_i} and switching back to the original partition. The execution of such an *interposed IRQ* is shown in Figure 5. In all three cases the IRQ queues are used, to prevent an out-of-order execution of IRQs.

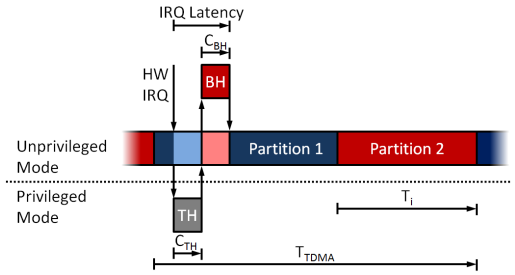


Figure 5: Interrupt Latency for interposed IRQ

For monitoring IRQs we use the δ^- based mechanism presented in [8] such that it permits bottom handler processing only with a defined minimum temporal distance d_{min} between any two consecutive activations (i.e. the parameter l in [8] for the length of the δ^--function is set to 1). I.e. if two interrupts are separated by at least d_{min} time units their respective bottom handler is directly executed after the top handler. However, if the second interrupt arrives less than d_{min} after the first, regular processing for delayed IRQs is used. A more complex setup with $l \geq 1$ is given in Appendix A.

5.1 Interference and Interrupt Latency

Next, we bound the worst-case interference interposed interrupt processing can cause for other partitions and develop a worst-case response time analysis for interposed interrupts. We start with the consideration of the interference on other partitions.

For our scenario interposed interrupts are permitted by the monitoring scheme every d_{min} time units. Thus, in any time interval of size Δt at most $\lceil \frac{\Delta t}{d_{min}} \rceil$ bottom-handler invocations may interrupt a partition. Each interposed interrupt executes for C_{BH_i} (enforced through the hypervisor). Furthermore, overhead for manipulation of the scheduler with WCET C_{sched} and two additional context switches with worst-case context-switch overhead of C_{ctx} each are introduced, which effectively increases the execution time of the bottom handler, i.e.

$$C'_{BH_i} = C_{BH_i} + C_{sched} + 2 \cdot C_{ctx} \tag{13}$$

Thus, the interference through interposed bottom-handler execution on other partitions is given through

$$I_{interposed} = \left\lceil \frac{\Delta t}{d_{min}} \right\rceil \cdot C'_{BH_i} \tag{14}$$

Furthermore, additional overhead is introduced through the modification of the top handler. The top handler calls the monitoring function for each interrupt that occurs during other partitions' timeslot. This effectively extends the worst-case execution time of the top handler to

$$C'_{TH_i} = C_{TH_i} + C_{Mon} \tag{15}$$

where C_{Mon} denotes the worst-case execution time of the monitoring function. Note, that neither (14) nor (15) depend on runtime behavior of a partition. I_p in (2) is now replaced by (14) which is strictly controlled by the hypervisor and the defined monitoring condition to maintain sufficient temporal independence for each partition.

Next, we regard the worst-case interrupt latencies. We distinguish between two scenarios. 1) all interrupts satisfy the monitoring condition given through d_{min} and 2) interrupts can arrive earlier than the specified d_{min}.

1) All interrupts that satisfy d_{min} do not have to wait until their respective partition is activated. Therefore, we can now remove $I_{TDMA}(W_i(q))$ from (7). Additional interference is introduced through the additional context switches through interposed interrupt processing (cf. (13)) and through the monitoring overhead in the top handlers (cf. (15)). With these modified WCETs and dropping the interference through the TDMA cycle we obtain from (11) for interrupts that adhere to d_{min}

$$W_i(q) = q \cdot C'_{BH_i} + \eta_i^+(W_i(q)) \cdot C'_{TH_i} + \sum_{j \in \mathcal{I}_i} \eta_j^+(W_i(q)) \cdot C_{TH_j} \tag{16}$$

2) Interrupts that violate monitoring condition are processed as delayed IRQs. No additional context switch is introduced (i.e. C_{BH_i} remains unchanged), however, the monitoring function is still executed in the top handler (i.e. C'_{TH_i} applies). Therefore (7) with C'_{TH_i} instead of C_{TH_i} applies.

From this analysis of worst-case timing effects we make the following observations. 1) Use of interposed interrupts only imposes bounded interference on other partitions. This interference is strictly controlled through the hypervisor and independent of the partition's timing behavior. 2) Worst-case interrupt latencies are independent of the TDMA cycle if interrupts arrive according to the specified d_{min}. 3) Interrupt latencies can be increased when they violate d_{min} as additional monitoring functionality is executed. This monitoring overhead is in the order of magnitude of 10 cycles [8] and, therefore, tolerable in most cases.

6. EVALUATION

In this section we evaluate our approach based on an own implementation in uC/OS-MMU running on a ARM926ejs processor @200MHz. We evaluate the approach w.r.t. average and worst-case interrupt latencies.

To evaluate our approach we will first focus on the IRQ latency improvements and secondly present the runtime and memory overhead.

6.1 IRQ latencies

In order to investigate IRQ latencies, we have performed two different experiments. The first experiment evaluates the interrupt latencies for the case when interrupts can arrive at arbitrary times, and thus can violate d_{min}. In the second experiment all interrupts adhere to d_{min}. A third testcase based on a measured activation trace from an automotive ECU is presented in Appendix A. In either scenario we trigger IRQs with one of the timers of the processor. The timer is reprogrammed from within the IRQ top handler such that the temporal distances between successive IRQs follow an exponential distribution with mean interarrival time λ. For the second scenario the pseudo-random interarrival time is set at least to d_{min} such that the monitoring condition is always satisfied. All interarrival times are generated before execution of the experiments in order not to introduce additional overhead in the top handler.

To measure the interrupt latencies (i.e. the time between top handler activation and finishing of the corresponding bottom handler), we use a second timer which provides a timestamp. Both, top handler and bottom handler, read this timestamp. The difference provides the measured IRQ latency.

We use a setup with two application partitions as shown in Figure 1. Each application partition has a TDMA slot length of $6000\mu s$. In addition to the two application partitions a third partition with slot length of $2000\mu s$ is used for housekeeping within the hypervisor. In our test setup we monitor the activation pattern of one IRQ source.

In order to evaluate the approach of interposed interrupts for a variety of long-term bottom-handler interrupt loads U_{IRQ}, we have varied the mean interarrival rate λ while keeping C'_{BH_i} constant, i.e. λ was set to

$$\lambda = C'_{BH_i} / U_{IRQ} \tag{17}$$

We have performed experiments for U_{IRQ} values of 1%, 5% and 10%. The results over 15000 IRQs are presented in Figure 6 (cumulative over all interrupt loads). The graphs show histograms over the IRQ latencies for the case without monitoring (Figure 6a), the case with monitoring (Figure 6b) and the monitored case where all interrupts adhere to d_{min} (Figure 6c). For the monitored scenarios we have used $\lambda = d_{min}$. Further, note that we have used a broken y-axis with dual scale for better readability.

In the scenario without monitoring (Figure 6a) we observe that direct IRQs ($\sim 40\% \widehat{=} 6000 IRQs$) exhibit a short latency of up to $50\mu s$. The interrupts that did not arrive in the appropriate time partition (i.e. delayed interrupts, 60%) have latencies of up to $T_{TDMA} - T_i = 8000\mu s$. The latencies of these interrupts are approximately uniformly distributed in this interval as the interrupts are activated at arbitrary points in the TDMA cycle. The vertical line in Figure 6a denotes the average IRQ latency of $\sim 2500\mu s$ over all 15000 IRQs.

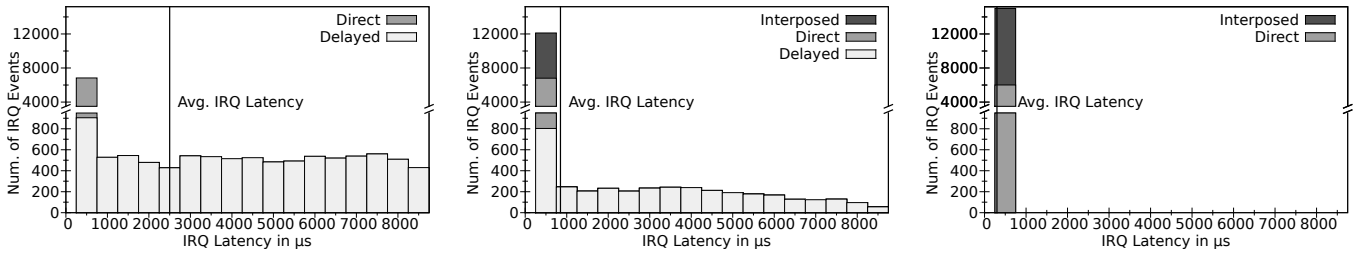
Figure 6b shows the results for the same activation pattern for the monitored case. With activated monitoring a significant amount of IRQs that were delayed in the previous scenario are now handled as interposed interrupts (direct 40%, interposed 40%, delayed 20%). The resulting interrupt latencies are significantly shorter. The average IRQ latency is reduced to $\sim 1200\mu s$. The worst-case interrupt latency is, as anticipated in Section 5, still defined through the TDMA cycle length and identical to the unmonitored case.

The results for the third scenario, where all IRQs satisfy d_{min}, is shown in Figure 6c. We observe that no IRQ is delayed (direct 40%, interposed 60%). The average IRQ latency is $\sim 150\mu s$, which marks a $\sim 16\times$ improvement over the unmonitored case. The worst-case latencies are no longer defined through the TDMA cycle length.

6.2 Memory and Runtime Overhead

Next, we evaluate the overhead on memory and computation time that the proposed interposed interrupt handling causes. The results for memory usage and runtime overhead are based on the same compiler optimization level (gcc with $-o1$).

The entire implementation requires 1120 bytes of code within the hypervisor. The modification of the hypervisor's TDMA scheduler account for 392 bytes, the implementation of the modified top handler (Figure 4b) for 456 bytes, and



(a) Monitoring disabled (b) Monitoring enabled (c) Monitoring enabled, no violations

Figure 6: Latency results for 15000 IRQs

the monitoring function for 272 bytes. The data memory overhead is caused solely by the used monitoring scheme and requires 28 bytes.

The runtime overhead is given through C_{Mon}, C_{sched} and two additional context switches of C_{ctx} each (cf. Section 5). C_{Mon}, i.e. the execution of time of the monitoring (including the instructions that call the scheduler if the IRQ handling is interposed), requires 128 instructions. The manipulation of the scheduler for processing interposed bottom handlers C_{sched} requires 877 instructions. The context switch overhead is heavily dependent on processor architecture (particularly cache and TLB) and memory layout. We measured an overhead of $\sim$ 5000 instructions per context switch for invalidation of caches and TLB on the ARMv5 architecture. Additional overhead based on cache writebacks of $\sim$ 5000 cycles per context switch was required for our particular memory layout. In scenario 2, for the given choice of $d_{min} = \lambda$ (which defines the number of interrupts that can be interposed) we have observed an overall increase in the number of context switches of $\sim$ 10%.

7. CONCLUSION

In this paper we have presented an approach to reduce interrupt latencies in hypervisor systems that provide sufficient temporal independence between partitions. We have provided an analysis for worst-case interrupt latencies for the case of strictly TDMA scheduled partitions. Furthermore we introduced a monitoring-based interrupt shaping, which permits the execution of IRQ bottom handlers within foreign TDMA slots up to a predefined density. We have shown that the monitoring ensures a sufficient temporal independence while improving IRQ latencies. Further, we have provided an analysis of interrupt latencies and interference on other partitions for the monitored case. In the evaluation, which is based on an implementation in uC/OS-MMU on an ARM926ejs processor, we have validated the results and shown that interrupt latencies can be significantly reduced in the average as well as the worst case.

Acknowledgement

This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within SMARTS project, ref. FCOMP-01-0124-FEDER-020536

8. REFERENCES

[1] MicroC/OS-MMU. Internet. http://www.embedded-office.com/en/products/uC-OS-MMU.html.

[2] B. Blackham, Y. Shi, and G. Heiser. Improving interrupt response time in a verifiable protected microkernel. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 323–336. ACM, 2012.

[3] J. Y. L. Boudec and P. Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer Verlag, 2001.

[4] G. Heiser and B. Leslie. The OKL4 Microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, pages 19–24. ACM, 2010.

[5] International Electrotechnical Commission. IEC61508 Ed.2 - Functional Safety of Electrical/Electronic/Programmable Safety-related Systems, 2008.

[6] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee. Task-aware virtual machine scheduling for I/O performance. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 101–110. ACM, 2009.

[7] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 201–209. IEEE, 1990.

[8] M. Neukirchner, T. Michaels, P. Axer, S. Quinton, and R. Ernst. Monitoring arbitrary activation patterns in real-time systems. In *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pages 293–302. IEEE, 2012.

[9] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling i/o in virtual machine monitors. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 1–10. ACM, 2008.

[10] P. Prisaznuk. ARINC 653 role in Integrated Modular Avionics (IMA). In *Digital Avionics Systems Conference (DASC 2008)*, 2008.

[11] J. Regehr and U. Duongsaa. Preventing interrupt overload. In *ACM SIGPLAN Notices*, volume 40, pages 50–58. ACM, 2005.

[12] K. Richter. *Compositional Scheduling Analysis Using Standard Event Models*. PhD thesis, Technical University of Braunschweig, Department of Electrical Engineering and Information Technology, 2004.

[13] Robert Kaiser, Stephan Wagner. Evolution of the PikeOS Microkernel. In *MIKES 2007 First International Workshop on Microkernels for Embedded Systems*, 2007.

[14] S. Ruocco. Real-time programming and L4 microkernels. In *Proceedings of the 2006 Workshop on Operating System Platforms for Embedded Real-Time Applications, Dresden, Germany*, 2006.

[15] S. Schliecker. *Performance analysis of multiprocessor real-time systems with shared resources*. PhD thesis, Technical University of Braunschweig, Department of Electrical Engineering and Information Technology, Braunschweig, Germany, 2011.

[16] S. Schliecker, J. Rox, M. Ivers, and R. Ernst. Providing accurate event models for the analysis of heterogeneous multiprocessor systems. In *Proc. 6th Int'l. Conf. on Hardware Software Codesign and System Synthesis (CODES-ISSS)*, 2008.

[17] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and microprogramming*, 40(2):117–134, 1994.

[18] P. Varanasi and G. Heiser. Hardware-supported virtualization on arm. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 11. ACM, 2011.

[19] J. Whiteaker, F. Schneider, and R. Teixeira. Explaining packet delays under virtualization. *ACM SIGCOMM Computer Communication Review*, 41(1):38–44, 2011.

APPENDIX

A. TESTCASE WITH A REAL-LIFE ACTIVATION PATTERN

To test the developed mechanism with a real-world example, we used a task activation trace from an automotive ECU with ~ 11000 activations. We assume that for each task activation in the trace, an IRQ for a partition on the hypervisor system is generated (e.g. the activated task communicates with a partition on the hypervisor system via CAN or Ethernet). Based on the activation trace, we generated a distance array which includes all distances between two consecutive activations in the trace. The generated array was now used to reload the timer within the IRQ top handler, instead of the exponential distributed distance array from Section 6.1.

We used the first 10% of the generated array to learn a $\delta^-_{I_p}[l]$ function with $l = 5$ entries. During this learning phase only delayed and direct IRQ handling is active. Therefore the average IRQ latency during this time is compareable to the scenario without monitoring in Section 6.1. For each IRQ Algorithm 1 is executed within the top handler. The algorithm checks if the distances between the current IRQ and the last l-events are smaller than the actual saved ones (i.e. $\delta^-_{I_p}[l]$ is initialized with large positive numbers). The current timestamp then is saved, if the current distance to the considered activation is smaller.

Algorithm 1 Self-learning $\delta^-_{I_p}[l]$ function

Input: timestamp, tracebuffer[l], $\delta^-_{I_p}[l]$
1: **for** $i \in [0, l-1]$ **do**
2: **if** timestamp − tracebuffer[i] $< \delta^-_{I_p}[i]$ **then**
3: $\delta^-_{I_p}[i]$ = timestamp − tracebuffer[i]
4: right shift tracebuffer
5: tracebuffer[0] = timestamp

When the learning phase is finished, Algorithm 2 compares the recorded $\delta^-_{I_p}[l]$ function to the predefined upper bound $\delta^-_{\widehat{I}_p}[l]$. If a distance in $\delta^-_{I_p}[l]$ is smaller than the counterpart in $\delta^-_{\widehat{I}_p}[l]$, the value in $\delta^-_{I_p}[l]$ is adjusted to adhere to $\delta^-_{\widehat{I}_p}[l]$.

Algorithm 2 Adjusting $\delta^-_{I_p}[l]$ to an upper bound

Input: $\delta^-_{\widehat{I}_p}[l]$, $\delta^-_{I_p}[l]$
1: **for** $i \in [0, l-1]$ **do**
2: **if** $\delta^-_{I_p}[i] < \delta^-_{\widehat{I}_p}[i]$ **then**
3: $\delta^-_{I_p}[i] = \delta^-_{\widehat{I}_p}[i]$

After processing all l elements, the system enters the monitored run mode and the remaining activations are processed. Now each IRQ within a foreign TDMA slot is interposed if it adheres to the generated and bounded $\delta^-_{I_p}[l]$ monitoring condition.

To test this setup we used four different predefined functions for $\delta^-_{\widehat{I}_p}[l]$. The results are shown in Figure 7. For the first test (graph a), an $\delta^-_{\widehat{I}_p}[l]$ was specified that does not bound the recorded $\delta^-_{I_p}[l]$. As expected the average IRQ latency drops from $\sim 2200\mu s$ to $\sim 120\mu s$, when entering the monitored run mode. Due to the fact, that the unmodified $\delta^-_{I_p}[l]$ is below $\delta^-_{\widehat{I}_p}[l]$, each IRQ within a foreign TDMA slot is interposed and none is delayed.

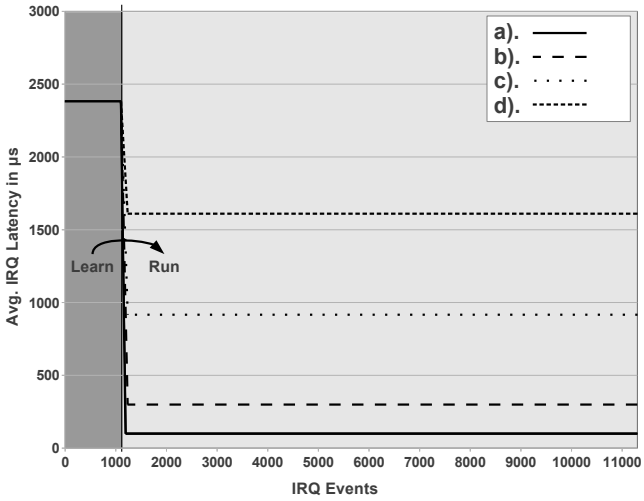


Figure 7: Average Interrupt Latency

For the second run (presented in graph b) we defined an $\delta^-_{\widehat{I}_p}[l]$ that only allows 25% of the requested load from the recorded $\delta^-_{I_p}[l]$. We did the same also for $12,5\%$ (presented in graph c) and for $6,25\%$ (presented in graph d). As expected bounding the requested load results in delayed IRQs, therefore we get as average latencies $\sim 300\mu s$ for 25%, $\sim 900\mu s$ for $12,5\%$ and $\sim 1600\mu s$ for $6,25\%$ allowed load. These results match to the shown synthetic testcases from Section 6.1.