Hardware-assisted interrupt delivery optimization for virtualized embedded platforms

Carlos Moratelli, Sergio Filho and Fabiano Hessel
Faculty of Informatics - PUCRS - Av. Ipiranga 6681, Porto Alegre, Brazil
Email: {carlos.moratelli, sergio.filho, fabiano.hessel}@.pucrs.br

***Abstract*—Virtualization is already a reality in modern embedded systems. Besides the direct relationship with cost reduction and improved resource utilization, virtualization enables the integration of real-time and general-purpose operating systems and applications on the same hardware platform. The resulting system may inherit deterministic timing characteristics for real-time along with a large software code base for general-purpose operating systems. However, the hypervisor must be carefully designed to take advantage of both types of operating systems. In this paper, we propose an interrupt policy for an embedded hypervisor using hardware-assisted virtualization. Our technique is flexible and can be adopted by applications with different timing constraints. Experimental results show that the interrupt delivery jitter on virtualized systems is close to non-virtualized when the proposed approach is used.**

I. Introduction

Virtualization technology allows the execution of several different operating systems (OS) on single and multi-core processors. A virtual machine (VM) is a software sandbox that acts as a physical computer executing a guest OS. The software layer responsible for VM's resource control is called virtual machine monitor (VMM) or hypervisor. Such VMM is one of the most important pieces of software regarding virtualization. Increasing demands for greater software integration have led the industry to adopt virtualization technology in embedded systems (ES). Nonetheless, ESs have different characteristics from the already well-stablished general-purpose virtualization. OSs are developed for different purposes resulting in RTOSs, general-purpose OSs or mobile OSs. In the same way, the development of hypervisors is also focused on different applications. For example, cloud virtualization usually privileges throughput as performed by Xen [1] and KVM [2]. On the other hand, ESs have characteristics that can be contradictory to cloud computing purposes, such as timing constraints, low CPU usage and small memory footprint. Thus, well-known hypervisors designed for general-purpose scenarios cannot be directly applied to ES due to the aforementioned limitations.

The diversity of ESes goals and requisites besides the complexity to adapt an existing hypervisor to attend specific needs led to the appearance of several embedded hypervisors with different purposes. For example, OKL4 [3] and SPUMONE [4] were designed for non-critical ES virtualization mixing general purpose OSs and RTOSs. MultiPARTS [5] and AUTOSTAR [6] were designed for automotive and critical embedded systems, respectively. Beyond that, the large variety of embedded processor families along their unique characteristics stimulate the appearance of hypervisors dedicated to specific platforms. For example, SPUMONE supports the SH4A processor while OKL4 was designed for the ARM architecture.

In this paper, we propose an interrupt policy for embedded hypervisors using the hardware-assisted virtualization module from the MIPS32 processor family. We have implemented the proposed technique in our hypervisor previously presented in [7]. Our main contribution is to show how the implementation of a hypervisor policy can take advantage of hardware-assisted virtualization to simplify the design while improving performance, in this case, of the interrupt subsystem. The remaining of the paper is organized as follows. Section II shows the main related work. Section III depicts our proposed technique. Section IV shows the experimental results. Finally, Section V describes our conclusion and presents a discussion about our findings.

II. Related Work

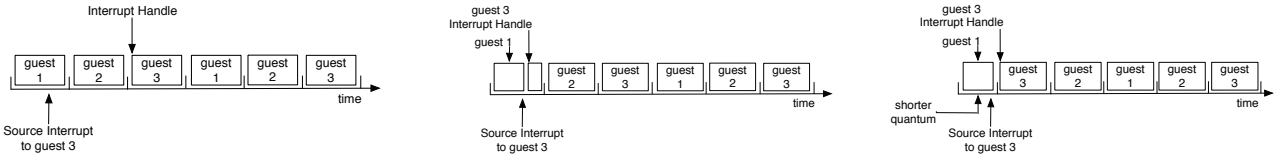
SPUMONE [4] is a small and lightweight virtualization layer designed to handle only the virtual CPU scheduling and dispatch of interrupts to the proper guest OS. Differently from most hypervisors, SPUMONE keeps the guest OSs in the processor's kernel mode. Thus, the guest OS can execute most of the sensitive or privileged instructions. However, a few sensitive instructions must be emulated using the para-virtualization approach. SPUMONE intercepts all interrupts, delivering them to specific guest OSs. That is possible because all interrupt registration routines in the guest OS are replaced by hypercalls. Despite SPUMONE's simplicity and low overhead, it presents some drawbacks: i) the para-virtualization technique requires modifications on the guest OS increasing the engineering efforts; ii) all interrupts must be investigated by SPUMONE, delaying the interrupt delivery. Our hypervisor overcomes SPUMONE's limitations since it allows full-virtualization and direct mapping for interrupts, while still keeping small footprint and low overhead.

Xvisor [8] is an embedded hypervisor able to support both full and para-virtualization. Similar to other embedded hypervisors, it aims to provide a lightweight layer with reduced overhead and small footprint. Full-virtualization is supported by the use of ARM virtualization extensions avoiding the need for guest OS modifications. Still, it can map interrupts directly to guests, allowing guest interrupt handling without the intervention of the hypervisor. However, Xvisor only supports para-virtualization on the MIPS 24k processor model under the Qemu emulator. In this case, direct mapped interrupts are not allowed. Our hypervisor was natively designed for the MIPS32 processor architecture with virtualization extensions. Still, we provide an interrupt policy to deal with interrupts asserted when the target guest OS is not executing. Finally, our hypervisor is currently supporting the SEAD-3 development board.

III. Fast interrupt delivery policy

Our hypervisor was designed with focus on hardware-assisted virtualization - more specifically, the MIPS M5150

978-1-5090-0246-7/15/$31.00 ©2015 IEEE



(a) Interrupt delayed. (b) Fast interrupt with recycled quantum. (c) Fast interrupt with reset quantum.

Fig. 1: Quantum scheduler scheme for interrupt delivery.

processor with virtualization extensions (VZ module). Among other features, the MIPS VZ module allows to map an interrupt source directly to a guest OS avoiding hypervisor intervention. This mechanism is known as *interrupt pass-through* and it allows to support directly mapped devices, i.e., not shared devices between guests, but directly accessed by a certain guest OS. The main advantage of this technique is low overhead, which is close to non-virtualized systems in terms of throughput and latency. However, if an interrupt occurs during the execution of any other guest OS, the hypervisor must decide between: i) keep the interrupt masked and delay its delivery or; ii) intercept the interrupt and reschedule the guest OSs. This allows the implementation of different policies for interrupt control. For example, for low priority devices such as a serial port for user console, the hypervisor may choose to keep the interrupt masked at supervisor mode. Thus, the guest OS will handle the interrupt only on its next execution (hypervisor scheduling quantum), resulting in a relatively long delay. That requires a large enough buffer queue in hardware or hardware flow control, otherwise, data may be lost. This arrangement is acceptable for low bandwidth devices. Devices with higher priority or higher bandwidth, on the other hand, may require hypervisor intervention to avoid long delays in the interrupt handling. In order to ilustrate a typical scenario, suppose that a Linux guest has an Ethernet device directly mapped. The hypervisor can keep the Ethernet interrupt unmasked aiming to perform a context switch between guests when a network packet is received, and the Linux guest is not executing.

Figure 1 depicts three guest OSs being scheduled in a round-robin fashion with three different quantum schemes for interrupt handling. Figure 1(a) shows an interrupt targeted to the guest OS 3 being asserted during the execution of the guest OS 1. Without a proper hypervisor policy, the interrupt delivery will be delayed until the execution of the guest OS 3. In fact, the overhead is similar to a non-virtualized system since there is not hypervisor intervention. However, this causes a long delay to deliver the interrupt. Our hypervisor was designed to make use of the interrupt pass-through mechanism, which allows the coexistence of general-purpose OSs and RTOSs with minimal interrupt delivery delay. First, we have studied two different schemes regarding the use of the scheduler quantum: recycle quantum and reset quantum. Figure 1(b) shows a fast interrupt delivery approach, causing the preemption of the current guest OS and the dispatch of the guest which will execute during the time remaining in the same quantum. In Figure 1(c), the current quantum is shortened, and the dispatched guest will execute during an entire new quantum. Section IV presents results and discussion about the implications of different approaches.

Independently of the quantum scheme adopted, our hypervisor accepts a set of rules to describe the behavior in front of different interrupt sources. Such set of rules is defined at design time. For each guest OS, the designers define which interrupts are directly mapped and, if such interrupt can induce the preemption of the current guest OS. Additionally, a higher priority guest OS can be marked to not be preempted at any circumstance. Table I shows a possible configuration for three guest OSs: two low priority Linux guests and a RTOS guest. The RTOS has the timer and serial 1 interrupts directly mapped. The column Root Int describes which interrupts will trigger the hypervisor when the guest is not executing. In this case, the hypervisor will intercept serial 1, dispatching the RTOS. Guest Linux 1 has the timer and serial 2 interrupts sources directly mapped. The hypervisor will intercept serial 2 interrupts when the guest is not executing only if the current guest is marked to be preempted. In the example, the hypervisor will preempt guest Linux 1 to delivery network interrupts to guest Linux 2, but the same interrupts will be delayed if the RTOS is executing. Thus, a high priority guest OS cannot be preempted by events from other guests, but only by the hypervisor scheduler. Our hypervisor provides a strong temporal isolation between guests, which is essential in ES virtualization if predictability of the RTOSs are a major concearn. Still, the flexibility of this approach allows the system to be configured for specific applications. Finally, our hypervisor always keeps the guest timer interrupt directly mapped, since it avoids hypervisor intervention during guest scheduling.

TABLE I: Example of a set of rules for a system configured with three guest OSs.

		Rules	
Guests	**Direct Mapped Interrupts**	**Root Int**	**Preempt**
RTOS	Timer, Serial 1	Serial 1	No
Linux 1	Timer, Serial 2	Serial 2	Yes
Linux 2	Timer, Network	Network	Yes

IV. Experimental Results

We conducted experiments on the SEAD-3 development platform board. The SEAD-3 supports the prototyping of MIPS processors allowing the user to evaluate the cores in an FPGA environment. The board can be used for performance benchmarking and software development. Our board is configured with the M5150 processor core running at 50MHz and 512Mbytes of external main memory. Still, the board has several peripherals including a Fast Ethernet (100Mbits/s) network device, two serial ports, USB, among others. In all experiments, we have used the Linux/MIPS (Linux port for MIPS architectures) release 4.0.0. Since our hypervisor performs full-virtualization, we used the same kernel binary for both virtualized and non-virtualized configurations. Still, we configured the hypervisor's scheduler to perform a round-robin scheduling algorithm.

Our experiments show how the interrupt handler response delay is affected due to hypervisor interference and how our proposed solution can improve the responsiveness. Therefore, we mapped a guest Linux to have direct access to the Ethernet

TABLE II: Average ($\overline{x}$), standard deviation (s), median (m) and 95^{th} percentile (p^{th}) for RTT of the messages in milliseconds.

Strategy	System															
	Native				2 VMs				4 VMs				6VMs			
	$\overline{x}$	s	m	p^{th}	$\overline{x}$	s	m	p^{th}	$\overline{x}$	s	m	p^{th}	$\overline{x}$	s	m	p^{th}
Without interrupt policy	1.073	0.107	1.05	1.24	9.35	10.21	2.73	29.5	36.33	30.70	29.6	86.4	66.29	49.40	66.9	145
Recycled quantum (R.Q.)	-	-	-	-	1.87	4.09	1.2	1.83	3.89	12.05	1.56	5.52	6.90	21.04	1.44	50.4
R.Q. with non-preempt. RTOS	-	-	-	-	9.28	10.16	2.70	29.1	7.41	12.90	1.66	29.1	9.72	19.83	1.68	31.5
Reset quantum (Res.Q.)	-	-	-	-	1.29	0.81	1.11	1.69	1.55	2.49	1.55	1.85	2.50	10.39	1.22	1.82
Res.Q. with non-preempt. RTOS	-	-	-	-	2.86	3.17	1.1	9.89	8.61	8.19	2.07	21.1	10.49	12.32	1.71	30.7

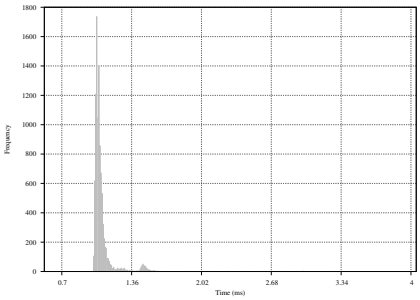


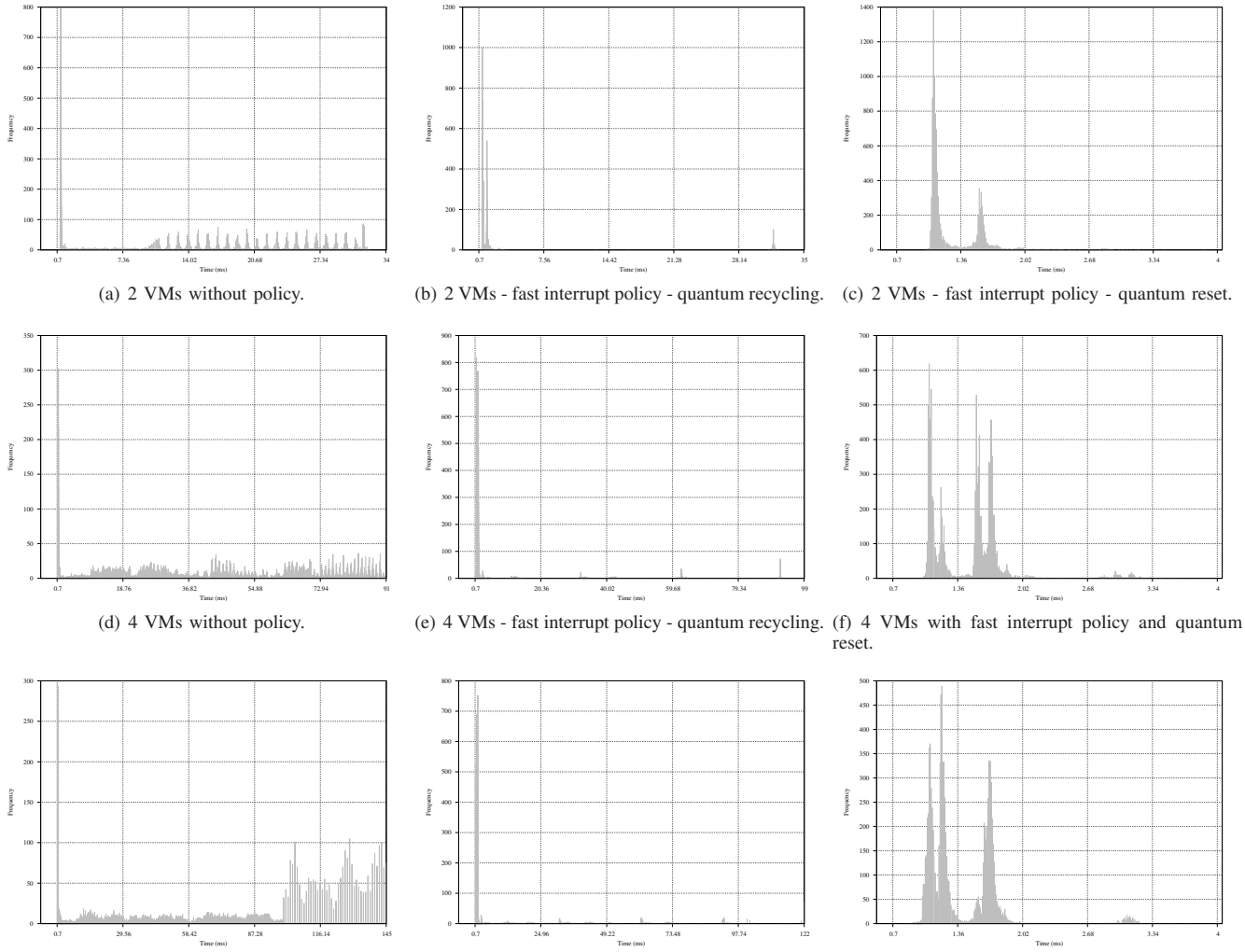
Fig. 2: Histogram for messages' RTT for native execution of the Linux/MIPS.

device. Thus, we used the ICMP (Internet Control Message Protocol) to send *echo request* messages from an Intel Xeon server running Debian Linux 7.8 directly connected to the SEAD-3 board. For all test cases, we sent ICMP messages at intervals of 50 milliseconds (20Hz). Each test case consists of 10,000 *echo request* messages followed by its respective response message (*echo reply*). Thus, it was determined the round-trip time (RTT) of the messages for Ethernet communication with the SEAD-3 board. Finally, we determined the average delay ($\overline{x}$), standard deviation (s), median (m) and the 95^{th} percentile (p^{th}) for messages' RTT. Table II shows the results for all experiments. Lines three and five depict the behavior of the recycled quantum and reset quantum policies with half of the VMs marked as non-preemptable, respectively. Results for the Linux/MIPS native (non-virtualized) execution can be viewed in the column named Native. Figure 2 shows the correspondent histogram for messages' RTT for native Linux execution. Figure 3 depicts the histogram for RTT of the messages for 2, 4 and 6 VMs without interrupt policy, fast interrupt policy with quantum recycling and quantum reset.

Once the non-virtualized response time was determined, we compared the results to virtualized instances of the Linux/MIPS. For each test case, the guest Linux/MIPS shared the processor with RTOSs in different system configurations: 2 VMs (one Linux/MIPS and one RTOS), 4 VMs (one Linux/MIPS and three RTOSs) and 6 VMs (one Linux/MIPS and five RTOSs). For each configuration, we performed tests without any specific policy, consequently increasing the response delay. Line one of the Table II shows that the delay growth is proportional to the increasing number of concurrent VMs. Figures 3(a), 3(d) and 3(g) show the corresponding histogram to the RTT of the messages to the configuration of 2, 4 and 6 VMs, respectively. Observe the increased horizontal scale of the histogram in the figures. These results show that the hypervisor heavily affected the response time when compared to native results. That was expected since the VMs were scheduled in a round-robin fashion, and an interrupt on the Ethernet device must wait for the next Linux/MIPS execution. Still, the hypervisor scheduler time quantum is 30 milliseconds. Thus, in a system configuration of 4 VMs the delay to handle an interrupt can be up to 90 milliseconds if the guest Linux/MIPS is the last of the round-robin queue.

After determining how the hypervisor affects the interrupt response time, we tested two different policies aiming to discover which one gives the best approximation to the native response time. Line two of Table II (Recycled quantum) shows the results when applied the fast interrupt policy together with the recycled quantum scheme (as explained in Figure 1(b)). This technique improves the overall results. However, the recycled quantum scheme has a drawback. If the remaining of quantum time is not sufficient to finish the interrupt handler routine, it will be finished only in the next guest's execution. Figures 3(b), 3(e) and 3(h) depict the corresponding histogram to the messages' RTT for the configuration of 2, 4 and 6 VMs, respectively. Most interrupts have a fast response as showed by the peak near 0.7 milliseconds. For such cases, the interrupt assertion happened during the guest execution or the remaining quantum time was enough to the interrupt handler finish its job. However, some messages experiment a long delay and the response time is near multiples of 30 milliseconds, i.e., the remaining quantum time was not enough and the interrupt handler only finished in the next execution. Line four of Table II (Reset quantum) shows the results for the fast interrupt policy with a reset quantum scheme. For 2 VMs configuration, the average and standard deviation delays are close to native execution. In the 4 and 6 VMs system configuration, the results are slightly higher, but still better than results for the recycled quantum scheme. Figures 3(c), 3(f) and 3(i) show the correspondent histogram for the ICMP echo reply delay for the configuration of 2, 4 and 6 VMs, respectively. Observe that the horizontal scale of the figure is similar to the native execution. The histograms show two peaks. The peak for delays under 1 millisecond represents interrupt assertions that happened when the target guest OS was in execution. The second peak for delays greater than 1 millisecond represents interrupt assertions that caused a context switch because the target guest was not in execution. Lines 3 (R.Q. with non-preempt. RTOS) and 5 (Res.Q. with non-preempt. RTOS) show results for recycled quantum and reset quantum schemes in the presence of non-preemptive guests, respectively. For each configuration, half of the guests was marked as non-preemptive. The presence of non-preemptive guests increased the average and standard deviation delays, but the results are still acceptable when compared the approach without any interrupt policy.

The reset quantum scheme associated with our policy rules presented better results. However, this scheme may be disruptive for RTOSs with strict timing constraints. The recycle quantum scheme keeps a strong time isolation between guests, being recommended when predictability is more important than low response time. Still, our fast interrupt policy allows the reduction of hypervisor interference to values near non-virtualized systems in certain configurations. Finally, our mechanism is flexible and allows to customize the hypervisor for specific applications for any number of guest OSs.



(a) 2 VMs without policy.
(b) 2 VMs - fast interrupt policy - quantum recycling.
(c) 2 VMs - fast interrupt policy - quantum reset.
(d) 4 VMs without policy.
(e) 4 VMs - fast interrupt policy - quantum recycling.
(f) 4 VMs with fast interrupt policy and quantum reset.
(g) 6 VMs without policy.
(h) 6 VMs - fast interrupt policy - quantum recycling.
(i) 6 VMs - fast interrupt policy - quantum reset.

Fig. 3: Histogram for messages' RTT for different system configurations.

V. Conclusion and discussion

Embedded processors with hardware-assisted virtualization enable the design of new hypervisors supporting full-virtualization. Still, specific embedded system constraints require configurable hypervisors that can be easily tuned for different applications. Our results showed that a hypervisor can be carefully designed to take advantage of hardware-assisted virtualization, resulting in average performance close to non-virtualized systems. Beyond the optimistic performance presented, our approach is flexible enough to support different embedded applications needs. As future work, we are planning to study the behavior of the different scheduler policies combined with our fast interrupt scheme.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003.

[2] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux Symposium*, vol. 1, Ottawa, Ontario, Canada, Jun. 2007.

[3] G. Heiser and B. Leslie, "The OKL4 microvisor: convergence point of microkernels and hypervisors," *APSys '10: Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, 2010.

[4] T. Lin, H. Mitake, and T. Nakajima, "Improving GPOS real-time responsiveness using vcpu migration in an embedded multicore virtualization platform," in *16th IEEE International Conference on Computational Science and Engineering, CSE 2013, December 3-5, 2013, Sydney, Australia*, 2013, pp. 693–700. [Online]. Available: http://dx.doi.org/10.1109/CSE.2013.107

[5] S. Trujillo, A. Crespo, and A. Alonso, "Multipartes: Multicore virtualization for mixed-criticality systems," in *Digital System Design (DSD), 2013 Euromicro Conference on*, Sept 2013, pp. 260–265.

[6] D. Reinhardt and G. Morgan, "An embedded hypervisor for safety-relevant automotive e/e-systems," in *Industrial Embedded Systems (SIES), 2014 9th IEEE International Symposium on*, June 2014.

[7] S. Zampiva, C. Moratelli, and F. Hessel, "A hypervisor approach with real-time support to the mips m5150 processor," in *Quality Electronic Design (ISQED), 2015 16th International Symposium on*, March 2015, pp. 495–501.

[8] A. Patel, M. Daftedar, M. Shalan, and M. Watheq El-Kharashi, "Embedded hypervisor xvisor: A comparative analysis," in *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*, March 2015, pp. 682–691.