Supporting Virtualization Standard for Network Devices in RTEMS Real-Time Operating System

Jin-Hyun Kim
Department of Smart ICT Convergence
Konkuk University
Seoul 143-701, Korea
jinhyun@konkuk.ac.kr

Sang-Hun Lee
Department of Computer Science and Engineering
Konkuk University
Seoul 143-701, Korea
shunlee@konkuk.ac.kr

Hyun-Wook Jin
Department of Computer Science and Engineering
Konkuk University
Seoul 143-701, Korea
jinh@konkuk.ac.kr

ABSTRACT

The virtualization technology is attractive for modern embedded systems in that it can ideally implement resource partitioning but also can provide transparent software development environments. Although hardware emulation overheads for virtualization have been reduced significantly, the network I/O performance in virtual machine is still not satisfactory. It is very critical to minimize the virtualization overheads especially in real-time embedded systems, because the overheads can change the timing behavior of real-time applications. To resolve this issue, we aim to design and implement the device driver of RTEMS for the standardized virtual network device called virtio. Our virtio device driver can be portable across different Virtual Machine Monitors (VMMs) because our implementation is compliant with the standard. The measurement results clearly show that our virtio can achieve comparable performance to the virtio implemented in Linux while reducing memory consumption for network buffers.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*real-time systems and embedded systems*

General Terms

Design, Performance

Keywords

Network virtualization, RTEMS, Real-time operating system, virtio, Virtualization

1. INTRODUCTION

The virtualization technology provides multiple virtual machines on a single device, each of which can run own operating system and applications over emulated hardware in an isolated manner [19]. The virtualization has been applied to large-scale server systems to securely consolidate different services with high system utilization and low power consumption. As modern complex embedded systems are also facing the size, weight, and power (SWaP) issues, researchers are trying to utilize the virtualization technology for temporal and spatial partitioning [5, 21, 12]. In the partitioned systems, a partition provides an isolated run-time environment with respect to processor and memory resources; thus, virtual machines can be exploited to efficiently implement partitions. Moreover, the virtualization can provide a transparent and efficient development environment for embedded software [10]. For example, if the number of target hardware platforms is smaller than that of software developers, they can work with virtual machines that emulate the target hardware system.

A drawback of virtualization, however, is the overhead for hardware emulation, which causes higher software execution time. Although the emulation performance of instruction sets has been significantly improved, the network I/O performance in virtual machine is still far from the ideal performance [13]. It is very critical to minimize the virtualization overheads especially in real-time embedded systems, because the overheads can increase the worst-case execution time and jitters, thus changing the timing behavior of real-time applications. Few approaches to improve the performance of network I/O virtualization in the context of embedded systems have been suggested, but these are either proprietary or hardware-dependent [7, 6].

In order to improve the network I/O performance, usually a paravirtualized abstraction layer is exposed to the device driver running in the virtual machine. Then the device driver explicitly uses this abstraction layer instead of accessing the original I/O space. This sacrifices the transparency of whether the software knows it runs on a real machine or a virtual machine, but can improve the network I/O performance avoiding hardware emulation. It is desirable to use the standardized abstraction layer to guarantee portability and reliability; otherwise, we would have to modify or newly implement the device driver for different Virtual Machine Monitors (VMMs) and have to manage different versions of device driver.

In this paper, we aim to design and implement the *virtio* driver for RTEMS [2], a Real-Time Operating System (RTOS) used in spacecrafts and satellites. virtio [17] is the standardized abstraction layer for paravirtualized I/O de-

EWiLi'15, October 8th, 2015, Amsterdam, The Netherlands.

vices and is supported by several well-known VMMs, such as KVM [8] and VirtualBox [1]. To the best of our knowledge, this is the first literature that presents detail design issues of the virtio front-end driver for RTOS. Thus, our study can provide insight into design choices of virtio for RTOS. The measurement results clearly show that our virtio can achieve comparable performance to the virtio implemented in Linux. We also demonstrate that our implementation can reduce memory consumption without sacrificing the network bandwidth.

The rest of the paper is organized as follows: In Section 2, we give an overview of virtualization and virtio. We also discuss related work in this section. In Section 3, we describe our design and implementation of virtio for RTEMS. The performance evaluation is done in Section 4. Finally, we conclude this paper in Section 5.

2. BACKGROUND

In this section, we give an overview of virtualization and describe virtio, the virtualization standard for I/O devices. In addition, we discuss the state-of-the-art for network I/O virtualization.

2.1 Overview of Virtualization and virtio

The software that creates and runs the virtual machines is called VMM or hypervisor. The virtualization technology is generally classified into full-virtualization and paravirtualization. The full-virtualization allows legacy operating system to run in virtual machine without any modifications. To do this, VMMs of full-virtualization usually perform binary translation and emulate every detail of physical hardware platforms. KVM and VirtualBox are examples of full-virtualization VMMs. On the other hand, VMMs of paravirtualization provide guest operating systems with programming interfaces, which are similar to the interfaces provided by hardware platforms but much simpler and lighter. Thus, the paravirtualization requires modifications of guest operating systems and can present better performance than full-virtualization. Xen [3] and XtratuM [12] are examples of paravirtualization VMMs.

virtio is the standard for virtual I/O devices. It was initially suggested by IBM [17] and recently became an OASIS standard [18]. The virtio standard defines paravirtualized interfaces between front-end and back-end drivers as shown in Fig. 1. The paravirtualized interfaces include two virtqueues to store send and receive descriptors. Because virtqueues are located in a shared memory between front-end and back-end drivers, the guest operating system and VMM can directly communicate each other without hardware emulation. Many VMMs, such as KVM, VirtualBox, and XtratuM, support virtio or its modification. General-purpose operating systems, such as Linux and Windows, implement the virtio front-end driver.

2.2 Related Work

There has been significant research on network I/O virtualization. The most of existing investigations are, however, focusing on the performance optimization for general-purpose operating systems [20, 15, 22, 16, 4]. Especially, the approaches that require supports from network devices are not suitable for embedded systems, because their network controllers are not equipped with sufficient hardware resources to implement multiple virtual network devices. Though there is an architectural research on efficient network I/O virtualization in the context of embedded systems [6], it also highly depends on the assist from network controller. The software-based approach for embedded system has been studied in a very limited scope that does not consider the standardized interfaces for network I/O virtualization [7].

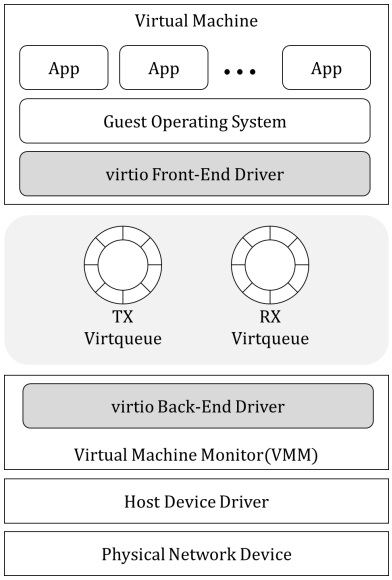

Figure 1: virtio.

Compared to existing research, virtio can be differentiated in that it does not require hardware support and can be more portable [17, 18]. The studies for virtio have mainly dealt with the back-end driver [11, 14]. However, there are several additional issues for the front-end driver on RTOS due to inherent structural characteristics of RTOS and the resource constraint of embedded systems. In this paper, we focus on the design and implementation issues of the virtio front-end driver for RTOS.

3. VIRTIO FOR RTEMS

In this section, we suggest the design of virtio front-end driver for RTEMS. Our design can efficiently handle hardware events generated by the back-end driver and mitigate memory consumption for network buffers. We have implemented the suggested design on the experimental system that runs RTEMS (version 4.10.2) over the KVM hypervisor as described in Section 4.1, but it is general enough to apply to other system setups.

3.1 Initialization

The virtio network device is implemented as a PCI device. Thus, the front-end driver obtains the information of the virtual network device through PCI configuration space. Once the registers of the virtio device are found in the configuration space, the driver can access the I/O memory of the virtio device by using the Base Address Register (BAR). The virtio header, which has the layout shown in Fig. 2, locates

Bits	Read / Write	Purpose
32	R	Device Features bits 0:31
32	R+W	Driver Features bits 0:31
32	R+W	Queue Address
16	R	Queue Size
16	R+W	Queue Select
16	R+W	Queue Notify
8	R+W	Device Status
8	R	ISR Status

Figure 2: virtio header in I/O memory.

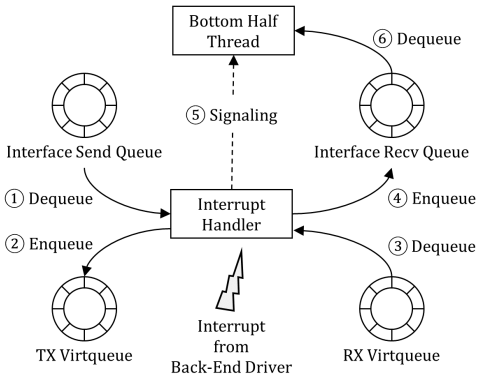

Figure 3: Hardware event handling.

in that I/O memory region and is used for initialization.

Our front-end driver initializes the virtio device through the virtio header as specified in the standard. For example, the driver decides the size of the virtqueues by reading the value in `Queue Size` region. Then the driver allocates the virtqueues in the guest memory area and lets the back-end driver know the base addresses of the virtqueues by writing these to the `Queue Address` region. Thus, both front-end and back-end drivers can directly access the virtqueues by means of memory referencing without expensive hardware emulation.

The front-end driver also initializes the function pointers of the general network driver layer of RTEMS with the actual network I/O functions implemented by the front-end driver. For example, the `if_start` pointer is initialized by the function that transmits a message through the virtio device. This function adds a send descriptor to the TX virtqueue and notifies it to the back-end driver. If the TX virtqueue is full, this function intermediately queues the descriptor to the interface queue described in Section 3.2.

3.2 Event Handling

The interrupt handler is responsible for hardware events. However, since the interrupt handler is expected to finish immediately relinquishing the CPU resources as soon as possible, the actual processing of hardware events usually takes place later. In general-purpose operating systems, such delayed event handling is performed by the bottom half that executes in interrupt context with a lower priority than the interrupt handler. In regard to network I/O, demultiplexing of incoming messages and handling of acknowledgment packets are the examples that the bottom half performs. However, RTOS usually do not implement a framework for bottom half; thus, we have to use a high-priority thread as a bottom half. The interrupt handler sends a signal to this thread to request the actual event handling, where there is a tradeoff between signaling overhead and size of interrupt handler. If the bottom half thread handles every hardware event aiming for a small interrupt handler, the signaling overhead can increase in proportional to the number of interrupts. For example, it takes more than 70 μs per interrupt in RTEMS for signaling and scheduling between a thread and an interrupt handler on our experimental system described in Section 4.1. On the other hand, if the interrupt handler takes care of most of events to reduce the signaling overhead, the system throughput can be degraded, because interrupt handlers usually disable interrupts during its execution.

In our design, the interrupt handler is only responsible for moving the send/receive descriptors between interface queues and virtqueues when the state of virtqueues changes. Fig. 3 shows the sequence of event handling, where the interface queues are provided by RTEMS and used to pass network messages between the device driver and upper-layer protocols. When a hardware interrupt is triggered by the back-end driver, the interrupt handler first checks if the TX virtqueue has available slots for more requests, and moves the send descriptor that is stored in the interface queue waiting for the TX virtqueue to become available (steps 1 and 2 in Fig. 3). Then the interrupt handler sees whether the RX virtqueue has used descriptors for incoming messages, and moves these to the interface queue (steps 3 and 4 in Fig. 3). Finally, the interrupt handler sends a signal to the bottom half thread (step 5 in Fig. 3) so that the actual processing for received messages can be processed later (step 6 in Fig. 3). It is noteworthy that the interrupt handler handles multiple descriptors at a time to reduce the number of signals. In addition, we suppress the interrupts with the aid from the back-end driver.

3.3 Network Buffer Allocation

On the sender side, the network messages are intermediately buffered in the kernel due to the TCP congestion and flow controls. As the TCP window moves, the buffered messages are sent as many as the TCP window allows. Thus, a larger number of buffered messages can easily fill the window size and can achieve higher bandwidth. On the receiver side, received messages are also buffered in the kernel until the destination task becomes ready. A larger memory space to keep the received messages also can enhance the bandwidth, because it increases the advertised window size in flow control. Although a larger TCP buffer size is beneficial for network bandwidth, the operating system limits the total size of messages buffered in the kernel to prevent the messages from exhausting memory resources. However, we have observed that the default TCP buffer size of 16 KByte in RTEMS is not sufficient to fully utilize the bandwidth provided by Gigabit Ethernet. Therefore, in Section 4.2,

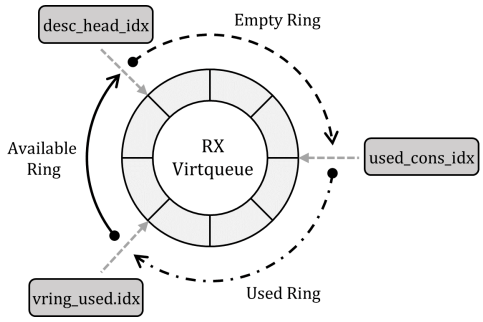

Figure 4: Controlling the number of preallocated receive buffers.

we heuristically search the optimal size of the TCP buffer that promises high bandwidth without excessively wasting memory resources.

Moreover, we control the number of preallocated receive buffers (i.e., *mbuf*). The virtio front-end driver is supposed to preallocate a number of receive buffers that matches the RX virtqueue, each of which occupies 2 KByte of memory. The descriptors of the preallocated buffers are enqueued at the initialization phase so that the back-end driver can directly place incoming messages in those buffers. This can improve the bandwidth by reducing the number of interrupts, but reserved memory areas can waste memory resources. Therefore, it is desirable to size the RX virtqueue based on the throughput of the front and back-end drivers. If the front-end driver can process more messages than the back-end driver, we do not need a large number of preallocated receive buffers. However, we need a sufficient number of buffers if the front-end driver slower than the back-end driver. The back-end driver of KVM requires 256 preallocated receive buffers, but we have discovered that 256 buffers are excessively large for Gigabit Ethernet as discussed in Section 4.2.

Fig. 4 shows how we control the number of preallocated receive buffers. The typical RX virtqueue has the used ring and available ring areas, which store descriptors for used and unused preallocated buffers, respectively. Our front-end driver introduces the empty ring area to the RX virtqueue in order to limit the number of preallocated buffers. At the initialization phase, we fill the descriptors with preallocated buffers until `desc_head_idx` reaches to the threshold defined as $sizeof(\text{virtqueue}) - sizeof(\text{empty ring})$. Then, whenever the interrupt handler is invoked, it enqueues the descriptors of new preallocated buffers as many as `vring_used.idx` − `used_cons_idx` (i.e., size of used ring). The descriptors in the used ring are retrieved by the interrupt handler as mentioned in Section 3.2. Thus, the size of empty ring is constant. We show the detail analysis of tradeoff between the RX virtqueue size and network bandwidth in Section 4.2.

4. PERFORMANCE EVALUATION

In this section, we analyze the performance of our design suggested in the previous section. First, we measure the impact of network buffer size on network bandwidth. Then, we compare the bandwidth and latency of our implementation with those of Linux.

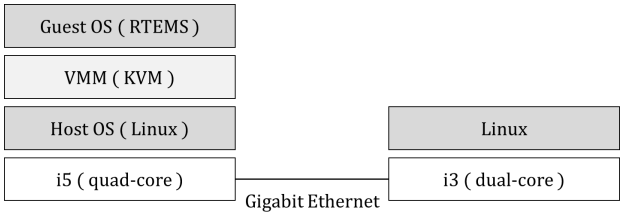

Figure 5: Experimental System Setup.

4.1 Experimental System Setup

We implemented the suggested design in RTEMS version 4.10.2. The mbuf space was configured in *huge* mode so that it is capable of preallocating 256 mbufs for the RX virtqueue. We measured the performance of the virtio on two nodes that were equipped with Intel i5 and i3 processors, respectively, as shown in Figure 5. The Linux version 3.13.0 was installed to the former, and we installed the Linux version 3.16.6 on the other node. The two nodes are connected directly through Gigabit Ethernet.

We ran the `ttcp` benchmarking tool to measure the network bandwidth with 1448 Byte messages, which is the maximum user message size that can fit into one TCP segment over Ethernet. We separately measured the send bandwidth and receive bandwidth of virtio by running a virtual machine only on the i5 node with KVM. We reported the bandwidth on the i3 node that ran Linux without virtualization. The intention behind such experimental setup is to measure the performance with the real timer because the virtual timer in virtual machines is not accurate [9].

4.2 Impact of Network Buffer Size

As mentioned in Section 3.3, we analyzed the impact of network buffer size on bandwidth. Fig. 6 shows the variation of send bandwidth with different sizes of the TCP buffer. We can observe that the bandwidth increases as the kernel buffer size increases. However, the bandwidth does not increase anymore after 68 KByte of TCP buffer size, because 68 KByte is sufficient to fill the network pipe of Gigabit Ethernet. Fig. 7 shows the experimental results for receive bandwidth, which also suggests 68 KByte as the minimum buffer size for the maximum receive bandwidth. Thus, we increased the TCP buffer size from 16 KByte to 68 KByte for our virtio.

We also measured the network bandwidth varying the size of the RX virtqueue as shown in Fig. 8. This figure shows that the network bandwidth increases until the virtqueue size becomes only 8. Thus, we do not need more than 8 preallocated buffers for Gigabit Ethernet though the default virtqueue size is 256.

In summary, we increased the in-kernel send and receive TCP buffer sizes from 16 KByte to 68 KByte, which requires additional memory resources of 104 KByte ($= (68-16) \times 2$) for higher network bandwidth. However, we reduced the number of preallocated receive buffers from 256 to 8 without sacrificing the network bandwidth, which saved 496 KByte ($= 256 \times 2 - 8 \times 2$) of memory, where the size of *mbuf* is 2 KByte as mentioned in Section 3.3. Thus, we saved 392

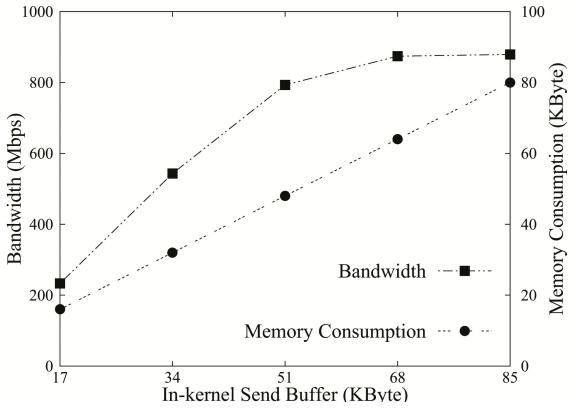

Figure 6: Tradeoff between TCP buffer size and send bandwidth.

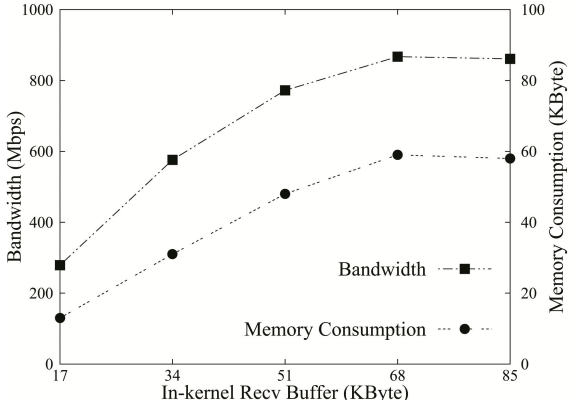

Figure 7: Tradeoff between TCP buffer size and receive bandwidth.

KByte ($= 496 - 104$) in total while achieving the maximum available bandwidth over Gigabit Ethernet.

4.3 Comparison with Linux

We compared the performance of RTEMS-virtio with that of Linux-virtio to see if our virtio can achieve comparable performance to the optimized one for general-purpose operating system. As shown in Fig. 9, the unidirectional bandwidth of RTEMS-virtio is almost the same with that of Linux-virtio, which is near the maximum bandwidth the physical hardware can provide in one direction. Thus, these results show that our implementation can provide quite reasonable performance with respect to bandwidth.

We also measured the round-trip latency in a way that two nodes send and receive the same size message in a ping-pong manner repeatedly for a given number of iterations. We reported the average, maximum, and minimum latencies for 10,000 iterations. Fig. 10 shows the measurement results for 1 Byte and 1448 Byte messages. As we can see in the figure, the average and minimum latencies of RTEMS-virtio are comparable to those of Linux-virtio. However, the maximum latency of RTEMS-virtio is significantly smaller than that of Linux-virtio (the y-axis is log scale) meaning RTEMS-virtio has a lower jitter. We always observed the

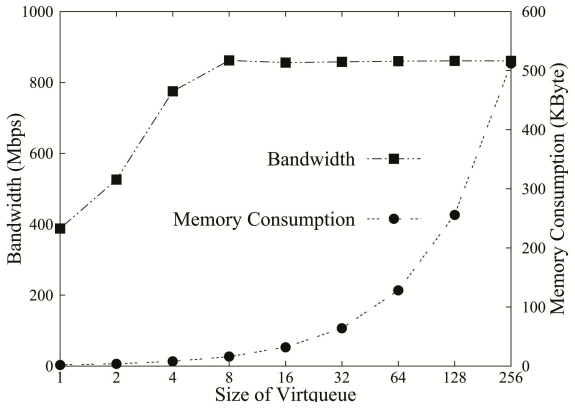

Figure 8: Tradeoff between RX virtqueue size and receive bandwidth.

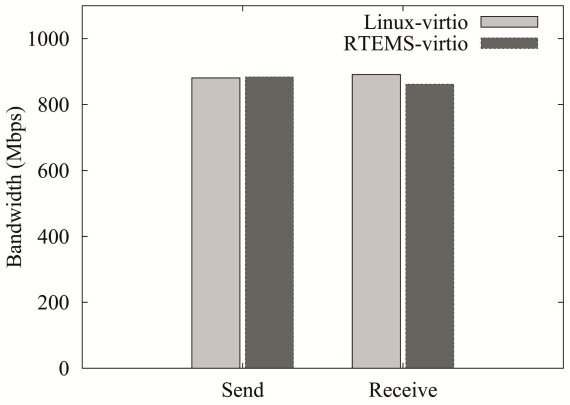

Figure 9: Comparison of bandwidth.

maximum latency in the first iteration of every measurement for both RTEMS and Linux. Thus, we presume that the lower maximum latency of RTMES is due to its smaller working set.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have suggested the design of the virtio front-end driver for RTEMS. The suggested device driver can be portable across different Virtual Machine Monitors (VMMs) because our implementation is compliant with the virtio standard. The suggested design can efficiently handle hardware events generated by the back-end driver and can reduce memory consumption for network buffers, while achieving the maximum available network bandwidth over Gigabit Ethernet. The measurement results have showed that our implementation can save 392 KByte of memory and can achieve comparable performance to the virtio implemented in Linux. Our implementation also has a smaller jitter of latency than Linux thanks to smaller working set of RTEMS. In conclusion, this study can provide insights into virtio from the viewpoint of the RTOS. Furthermore, the discussions in this paper can be extended to apply to other RTOS running in virtual machine to improve the network performance and portability.

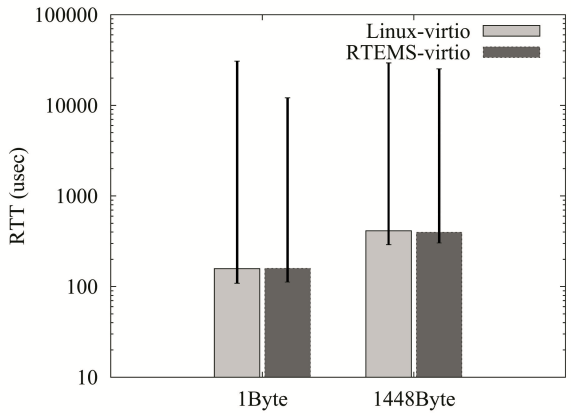

Figure 10: Comparison of latency.

As future work, we plan to measure the performance of our virtio on a different VMM, such as VirtualBox, to show that our implementation is portable across different VMMs. Then we will release the source code. In addition, we intend to extend our design for dynamic network buffer sizing and measure the performance on real-time Ethernet, such as AVB.

6. ACKNOWLEDGMENTS

This research was partly supported by the National Space Lab Program (#2011-0020905) funded by the National Research Foundation (NRF), Korea, and the Education Program for Creative and Industrial Convergence (#N0000717) funded by the Ministry of Trade, Industry and Energy (MOTIE), Korea.

7. REFERENCES

[1] *Oracle VM VirtualBox*. http://www.virtualbox.org.

[2] *RTEMS Real Time Operating System (RTOS)*. http://www.rtems.org.

[3] 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. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[4] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan. High performance network virtualization with sr-iov. *Journal of Parallel and Distributed Computing*, 72(11):1471–1480, 2012.

[5] S. Han and H.-W. Jin. Resource partitioning for integrated modular avionics: comparative study of implementation alternatives. *Software: Practice and Experience*, 44(12):1441–1466, 2014.

[6] C. Herber, A. Richter, T. Wild, and A. Herkersdorf. A network virtualization approach for performance isolation in controller area network (can). In *20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[7] J.-S. Kim, S.-H. Lee, and H.-W. Jin. Fieldbus virtualization for integrated modular avionics. In *16th IEEE Conference on Emerging Technologies & Factory Automation (ETFA)*, 2011.

[8] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Linux Symposium*, pages 225–230, 2007.

[9] S.-H. Lee, J.-S. Seok, and H.-W. Jin. Barriers to real-time network i/o virtualization: Observations on a legacy hypervisor. In *International Symposium on Embedded Technology (ISET)*, 2014.

[10] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[11] M. Masmano, S. Peiró, J. Sanchez, J. Simo, and A. Crespo. Io virtualisation in a partitioned system. In *6th Embedded Real Time Software and Systems Congress*, 2012.

[12] M. Masmano, I. Ripoll, A. Crespo, and J. Metge. Xtratum: a hypervisor for safety critical embedded systems. In *11th Real-Time Linux Workshop*, pages 263–272, 2009.

[13] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in xen. In *USENIX Annual Technical Conference*, pages 15–28, 2006.

[14] G. Motika and S. Weiss. Virtio network paravirtualization driver: Implementation and performance of a de-facto standard. *Computer Standards & Interfaces*, 34(1):36–47, 2012.

[15] H. Raj and K. Schwan. High performance and scalable i/o virtualization via self-virtualized devices. In *ACM Symposium on High-Performance Parallel and Distributed Computing*, June 2007.

[16] K. K. Ram, J. R. Santos, Y. Turner, A. L. Cox, and S. Rixner. Achieving 10gbps using safe and transparent network interface virtualization. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, March 2009.

[17] R. Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[18] R. Russell, M. S. Tsirkin, C. Huck, and P. Moll. *Virtual I/O Device (VIRTIO) Version 1.0*. OASIS, 2015.

[19] L. H. Seawright and R. A. MacKinnon. Vm/370 - a study of multiplicity and usefulness. *IBM Systems Journal*, 18(1):4–17, 1979.

[20] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference*, June 2001.

[21] S. H. VanderLeest. Arinc 653 hypervisor. In *29th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, 2010.

[22] L. Xia, J. Lange, and P. Dinda. Towards virtual passthrough i/o on commodity devices. In *Workshop on I/O Virtualization (WIOV)*, December 2008.