Virtio Front-End Network Driver for RTEMS Operating System

Jin-Hyun Kim and Hyun-Wook Jin, *Member, IEEE*

***Abstract*—Virtualization technology can provide a transparent and efficient software development environment at the development phase of embedded systems. In addition, virtualization can be exploited to provide strong resource isolation between real-time applications. However, since virtualization incurs noticeable overheads, we need to minimize these to utilize virtualization technology for either a software development environment or a runtime environment of real-time embedded systems. In this letter, we particularly focus on virtualization overheads of network I/O and present an implementation of standardized (i.e., virtio) front-end network driver for a real-time operating system (RTOS). Although there have been several studies for virtio on general-purpose OS, our implementation is the first open-source virtio for an RTOS. The measurement results show that our implementation can improve the bandwidth and latency up to 168% and 52%, respectively. In addition, the memory usage can be saved up to 36%.**

***Index Terms*—Network, real-time operating systems (RTOSs), virtio, virtualization.**

I. Introduction

As THE cost of implementing the hardware platforms for mission-critical systems, such as satellites and aircraft, is very expensive, it is difficult to provide enough hardware platforms for software developers at the development phase. Virtualization technology that provides virtual hardware platforms can resolve these problems by efficiently building a software development environment [1], [2]. Moreover, there are active movements to utilize virtualization technology in runtime software platforms of mission-critical systems. For example, virtualization is considered an attractive approach to implement temporal and spatial partitioning for avionics systems [3]. However, virtualization increases the execution time of applications and adds complexity to the analysis for worst-case execution time (WCET) and end-to-end delay. It is a well-known fact that virtualization increases particularly the overheads to access I/O peripherals, such as network and storage devices [4], [5]. Therefore, it is important to reduce virtualization overheads for both development and runtime environments.

The para-virtualized I/O standard called *virtio* [6], [7] can reduce the I/O overheads in virtualized environments, but it has not been implemented yet for a real-time operating system (RTOS). In this letter, we implement a virtio network driver for Real-Time Executive for Multiprocessor Systems (RTEMS) [8] aiming at reducing virtualization overhead of network I/O. RTEMS is a prevalent RTOS in mission-critical systems, such as space flight and medical devices; thus, providing a virtual platform with less overhead is especially important for RTEMS and its applications. To make use of virtio in an RTOS, we must provide a virtio driver that observes the device driver model of the RTOS. In addition, the driver should be designed in consideration of resource constraint of embedded systems. Our implementation is the first open-source implementation of the virtio network driver for an RTOS. Our virtio can support different CPU architectures and multiple hypervisors, such as Quick EMUlator (QEMU) [9], Kernel-based Virtual Machine (KVM) [10], and VirtualBox [11], and provide the dynamic buffer allocation for less memory requirements.

II. Background

A. Related Work

Virtualization technologies enable a single physical machine to run multiple virtual machines (VMs), each of which can run own OS. However, virtualization increases the runtime overheads especially for I/O operations due to additional software layer [4], [5]. In para-virtualization hypervisors, such as Xen [12], researchers tried to optimize the data path [13], buffer management [14], and scheduling of VMs [15] in terms of networking. In full-virtualization hypervisors, researchers suggested para-virtualized network devices to overcome inherent limitations of full-virtualization [6], [7]. Contemporary high-end network interfaces also provide the self-virtualization feature [16]. In this letter, we target the para-virtualized network device for full-virtualization hypervisors. We particularly aim to implement the front-end driver for an RTOS. The *front-end* driver is the device driver installed in the guest OS; the *back-end* driver resides in the hypervisor and is responsible for accessing the physical device. This letter is differentiated from our precursory work [17] in that we support multiple hypervisors and CPU architectures, provide dynamic network buffer allocation, and open the source codes publicly.

Manuscript received November 4, 2019; accepted December 1, 2019. Date of publication December 3, 2019; date of current version August 27, 2020. This work was supported by the National Research Foundation of Korea (NRF) funded by the Korea Government (MSIT) under Grant 2017R1A2B4012759. This manuscript was recommended for publication by H. Tomiyama. *(Corresponding author: Hyun-Wook Jin.)*

J.-H. Kim is with the Department of Electronic Warfare, LIG Nex1 Company, Ltd., Yongin 16911, South Korea (e-mail: jinhyun.kim@lignex1.com).

H.-W. Jin is with the Department of Computer Science and Engineering, Konkuk University, Seoul 05029, South Korea (e-mail: jinh@konkuk.ac.kr).

Digital Object Identifier 10.1109/LES.2019.2957570

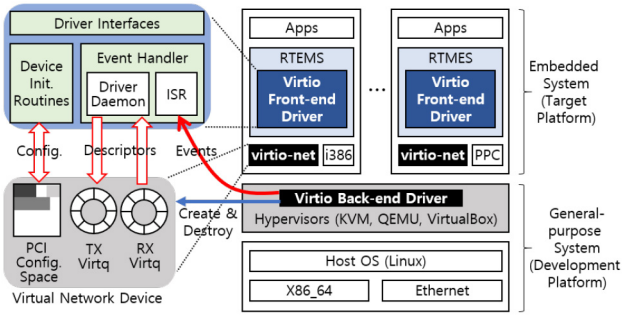

Fig. 1. Overall design of virtio network driver for RTEMS.

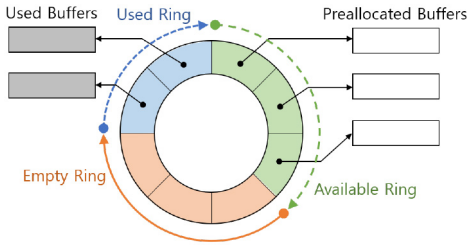

Fig. 2. Management of RX virtqueue.

B. Virtio

The hypervisors that provide full-virtualization, such as QEMU, KVM, and VirtualBox, do not require modifications to the software running on VMs including OS kernels. Full-virtualization can provide the transparency for existing software, but incurs high runtime overheads to emulate every detail of physical hardware. The virtio is the standardized para-virtualized interfaces between front-end drivers and back-end drivers [6], [7]. Since these para-virtualized interfaces of virtio remove the emulation overheads in the I/O path, we can achieve a significant performance improvement. The para-virtualized interfaces include the virtual PCI configuration space and two I/O queues called *TX Virtqueue* and *RX Virtqueue* that store send and receive descriptors, respectively. The I/O queues are in a shared memory area between the front-end and back-end drivers; thus, the guest OS and the hypervisor can directly communicate with each other without hardware emulation. Most general-purpose OS and full-virtualization hypervisors provide the virtio front-end and back-end drivers, respectively. However, contemporary RTOS do not implement yet the virtio front-end driver; thus, they cannot benefit from the para-virtualized I/O interfaces.

III. Virtio for RTEMS

In this section, we describe the design of the virtio front-end network driver for RTEMS. Our implementation is compatible with QEMU, KVM, and VirtualBox and operates on different CPU architectures, such as i386 and PowerPC (PPC). This is the first open-source implementation of virtio for an RTOS. The source code and instructions are available at https://github.com/sslab-konkuk/RTEMS-virtio/. The patch information also has been posted to the RTEMS mailing list.

A. Overview

Fig. 1 depicts the overall design of the virtio front-end driver for RTEMS, which comprises driver interfaces, device initialization routines, and event handler as follows.

Driver Interfaces: Provide interface functions to upper layers (i.e., TCP/IP suite), so that the networking applications and TCP/IP suite can utilize transparently the virtio. The internal implementation of the interfaces conforms to the network device driver model of RTEMS.

Device Initialization Routines: Read the virtual PCI configuration space to identify the virtio device provided by the back-end driver, perform initialization of the virtual device by allocating TX and RX Virtqueues, and register the event handler to the RTEMS kernel.

Event Handler: Consists of the interrupt service routine (ISR) and the driver daemon, which take care of top-half and bottom-half interrupt handling, respectively. In particular, the driver daemon is responsible for dynamic network buffer allocation described in Section III-B.

We implemented the virtio front-end driver based on that in FreeBSD, because RTEMS also borrowed the basic framework of TCP/IPv4 stacks from legacy FreeBSD. We newly added about 1000 lines of code and touched about 3000 lines of code, which is a significant implementation work at the OS level. Our implementation also follows the coding rules defined by the RTEMS community.

B. Dynamic Buffer Allocation

The contemporary device drivers generally allocate several network buffers and preregister these buffers to the RX queue when the network device is initialized. This is to immediately move the received messages from the network device to the host memory. If the preallocated network buffers are not provided, the network device must generate an additional interrupt per message to request the memory area to place the received message.

Existing virtio front-end drivers for general-purpose OS also register the preallocated network buffers during initialization. For instance, the RX Virtqueue in FreeBSD is managed by three indices and is divided into three areas named *available ring*, *used ring*, and *empty ring* as shown in Fig. 2. The available ring area has the descriptors for preallocated buffers. The virtio front-end driver in FreeBSD initializes the whole RX Virtqueue into available ring area (i.e., full of preallocated buffers). The used ring includes the pending buffers where received messages are placed but are not processed yet by the front-end driver. When a receive completion interrupt occurs, the event handler of the front-end driver dequeues the descriptors in the used ring. The queue slots, the descriptors of which were dequeued, belongs to the empty ring area. Thus, the event handler should reregister other preallocated network buffers to the empty ring for later use.

As the number of preallocated network buffer increases, the number of messages that can be moved in a batched manner without handshaking with the front-end driver increases, thereby achieving a higher network bandwidth. However, these preallocated network buffers consume significant memory resources. For instance, when the bandwidth required by an

Algorithm 1 Dynamic Buffer Allocation

```
1:  nmsg ← 0
2:  while used_ring ≠ EMPTY and nmsg ≤ MAX_BAND do
3:    desc ← RxVq_Dequeue()
4:    RxIf_Enqueue(desc)
5:    new_desc ← Alloc_Net_Buf()
6:    RxVq_Enqueue(new_desc)
7:    nmsq ← nmsg + 1
8:  end while
9:  if nmsg > available_limit then
10:   for nmsg − available_ring do
11:     new_desc ← Alloc_Net_Buf()
12:     RxVq_Enqueue(new_desc)
13:   end for
14:   available_limit ← nmsg
15: end if
```

application is low, the preallocated buffers waste the memory resources without any benefit. In embedded systems where hardware resources are limited, memory resources should be efficiently used. To save memory resources without sacrificing network bandwidth, our virtio front-end driver adjusts the number of preallocated buffers to the application workload. The main idea is to preallocate only few network buffers at the initialization phase and gradually increase the number of preallocated buffers as in Algorithm 1.

The above algorithm depicts the dynamic memory allocation performed by the driver daemon. In this algorithm, the while loop gets messages from the used ring (line 3), dispatches these messages to the upper layer (line 4), and register new preallocated buffers to the empty ring (lines 5 and 6). *nmsg* in line 7 represents the number of network messages processed within the while loop, and `MAX_BAND` is the maximum number of messages allowed to be processed by the driver daemon per interrupt. Lines 9–15 perform the dynamic network buffer allocation. The *available_limit* variable is initialized with a small value to limit the maximum number of preallocated network buffers (i.e., the size of the available ring) and gradually adjusted as the workload changes. The *nmsg* variable is considered the bandwidth required. The *nmsg* can be larger than *available_limit* (line 9), because a new network buffer is preallocated by lines 5 and 6 whenever a message is processed, while the back-end driver also keeps inserting the arrived messages to the RX Virtqueue. Thus, if *nmsg* is larger than *available_limit*, the algorithm increases the maximum number of preallocated buffers. Therefore, `MAX_BAND` and *available_limit* can control the maximum bandwidth available and the maximum memory usage. In the current implementation, we increase *available_limit* whenever the driver experiences a buffer shortage as shown in line 14. Thus, once *available_limit* converges with an enough runtime, there is no performance penalty due to dynamic buffer allocation.

C. Architectural Independence

In the board support package (BSP) of the virtio driver, we try to use architecture-independent functions as much as possible, so that our front-end driver can support multiple hypervisors and different CPU architectures. For example, we use the `pci_read_config()` function to access the virtual PCI configuration space provided by hypervisors. We also use `inport_byte()` and `outport_byte()` to read and write data from/to the virtual I/O space shared between the guest domain and the hypervisor. In addition, we avoid the architectural dependency in memory barrier by using the built-in function of the gcc compiler [i.e., `__sync_synchronize()`], instead of assembly instructions.

IV. Performance Evaluation

In this section, we measure the networking performance and memory usage of our virtio. The performance evaluation was conducted on two PCs equipped with an Intel Core i5 processor and an Intel Core i3 processor, respectively. Two nodes were directly connected through Gigabit Ethernet without a switch. We applied virtualization only to the i5-based node, while reporting performance numbers on the other node that was not virtualized, because the hypervisors do not provide an accurate timer to guest domains.

A. Comparisons With Full-Virtualized Drivers

We compared the performance of our virtio driver with that of legacy full-virtualized drivers on different hypervisors. For legacy drivers, we used the n2k driver for the ne2000 network controller virtualized by QEMU and the fxp driver for the i82559er network controller virtualized by KVM. Since we could not find an RTEMS network driver that successfully operated on VirtualBox, only the performance of our virtio was reported in the VirtualBox case. We ran RTEMS (version 4.11) as the guest OS on the PPC and i386-based VMs provided by QEMU, KVM, and VirtualBox hypervisors. The hypervisors ran on Ubuntu 14.04. We measured the round-trip latency and the bidirectional bandwidth by using ping-pong communication and burst transmission microbenchmarks, respectively. In our measurements, we used a relatively large message size of 1 KB to obtain the peak bandwidth achievable and to analyze the latency affected by the extra message copy of hypervisors. Regarding the parameters of socket buffers, we set `mbuf_bytecount` and `mbuf_cluster_bytecount` to 128 KB and 512 KB, respectively.

As we can see in Table I, the virtio driver could improve the latency and the bandwidth up to 52% and 168%, respectively. Moreover, it could save the memory consumed by the network buffers up to 36%. The less memory usage of virtio is owing to the dynamic buffer allocation described in Section III-B. These experiment results also show that our virtio front-end driver is compatible with different hypervisors and CPU architectures.

B. Comparisons With Existing Virtio Implementations

We also compared the performance of different virtio implementations. We used KVM in these experiments. As shown in Table II, our implementation reported comparable round-trip latency and bidirectional bandwidth to virtio drivers of Linux

TABLE I
COMPARISONS WITH LEGACY FULL-VIRTUALIZED DRIVERS

		Round-Trip Latency (μs)	Bi-directional Bandwidth (Mbps)	Memory Usage (KB)
PPC	Legacy	832	87	128
over	Virtio	401	233	106
QEMU	Imp.(%)	52	168	17
i386	Legacy	444	1424	36
over	Virtio	411	1611	23
KVM	Imp.(%)	7	13	36
i386	Legacy	-	-	-
over	Virtio	322	1033	144
VB	Imp.(%)	-	-	-

TABLE II
COMPARISONS WITH EXISTING VIRTIO IMPLEMENTATIONS

	Linux Virtio	FreeBSD Virtio	RTEMS Virtio
Round-Trip Latency (μs)	408	387	411
Bi-directional Bandwidth (Mbps)	1612	1604	1611

and FreeBSD. The virtio implementations of these general-purpose OS are very mature; thus, these experiment results confirm that our virtio provides quite reasonable performance.

V. CONCLUSION

In this letter, we presented an implementation of virtio front-end driver for RTEMS. Our implementation is compatible with several hypervisors and support different CPU architectures. The performance measurement results showed that, compared with legacy full-virtualized drivers, our virtio could improve the latency and the bandwidth up to 52% and 168%, respectively. Moreover, it could save the memory consumed by the network buffers up to 36%. We also showed that our virtio had comparable performance to virtio drivers of general-purpose OS. As future work, we plan to extend our virtio implementation to support advanced features, such as TCP segmentation offload (TSO) and generic segmentation offload (GSO).

REFERENCES

[1] P. S. Magnusson *et al.*, "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.
[2] H. P. Breivold and K. Sandström, "Virtualize for test environment in industrial automation," in *Proc. IEEE Emerg. Technol. Factory Autom. (ETFA)*, Barcelona, Spain, 2014, pp. 1–8.
[3] S. Han and H.-W. Jin, "Resource partitioning for integrated modular avionics: Comparative study of implementation alternatives," *Softw. Pract. Exp.*, vol. 44, no. 12, pp. 1441–1466, 2014.
[4] L. Cherkasova and R. Gardner, "Measuring CPU overhead for I/O processing in the Xen virtual machine monitor," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf. (ATC)*, 2005, p. 24.
[5] V. Chadha, R. Illiikkal, R. Iyer, J. Moses, D. Newell, and R. J. Figueiredo, "I/O processing in a virtualized platform: A simulation-driven approach," in *Proc. 3rd Int. Conf. Virtual Execution Environ. (VEE)*, San Diego, CA, USA, 2007, pp. 116–125.
[6] R. Russell, "Virtio: Towards a de-facto standard for virtual I/O devices," *ACM SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 95–103, 2008.
[7] R. Russell, M. S. Tsirkin, C. Huck, and P. Moll, *Virtual I/O Device (VIRTIO) Version 1.0*, OASIS, Burlington, MA, USA, 2015.
[8] OAR Corporation. *RTEMS: An Open Real-Time Operating System*. Accessed: Sep. 19, 2019. [Online]. Available: www.rtems.com
[9] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf. (ATC)*, 2005, p. 41.
[10] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The Linux virtual machine monitor," in *Proc. Linux Symp.*, 2007, pp. 225–230.
[11] ORACLE. *VirtualBox*. Accessed: Sep. 19, 2019. [Online]. Available: www.virtualbox.org
[12] P. Barham *et al.*, "Xen and the art of virtualization," *ACM SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, 2003.
[13] J. Liu, W. Huang, B. Abali, and D. K. Panda, "High performance VMM-bypass I/O in virtual machines," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf. (ATC)*, 2006, p. 3.
[14] A. Menon, A. L. Cox, and W. Zwaenepoel, "Optimizing network virtualization in Xen," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf. (ATC)*, 2006, p. 2.
[15] C.-H. Hong, K. Lee, H. Park, and C. Yoo, "ANCS: Achieving QoS through dynamic allocation of network resources in virtualized clouds," *Sci. Program.*, vol. 2016, May 2016, Art. no. 4708195.
[16] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High performance network virtualization with SR-IOV," *J. Parallel Distrib. Comput.*, vol. 72, no. 11, pp. 1471–1480, 2012.
[17] J.-H. Kim, S.-H. Lee, and H.-W. Jin, "Supporting virtualization standard for network devices in RTEMS real-time operating system," *ACM SIGBED Rev.*, vol. 13, no. 1, pp. 35–40, 2016.