IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 23, 2018, accepted September 26, 2018, date of publication October 22, 2018,
date of current version December 18, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2877296

Achieving Consistent Real-Time Latency at Scale in a Commodity Virtual Machine Environment Through Socket Outsourcing-Based Network Stacks

OSCAR F. GARCIA[1], YASUSHI SHINJO[1], (Member, IEEE), AND CALTON PU[2], (Fellow, IEEE)
[1]Department of Computer Science, University of Tsukuba, Tsukuba 305-8573, Japan
[2]College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA

Corresponding author: Oscar F. Garcia (oscar@softlab.cs.tsukuba.ac.jp)

This work was supported by JSPS KAKENHI under Grants 16K12410 and 25540022.

ABSTRACT It is challenging to achieve a consistent real-time (RT) response time in commodity virtual machine (VM) environments because they have longer and more complex network protocol stacks. This paper analyzes such network stacks and proposes a method that achieves consistent latency in a Linux KVM-based hosted environment. The analysis identifies a priority inversion in the interrupt-first host kernel of vanilla Linux, and the proposed method addresses it by using the PREEMPT_RT patch. Subsequently, the analysis identifies another priority inversion in softirq handling of the host kernel. The proposed method addresses it by dividing softirq handling into RT and non-RT types. The analysis then identifies the cache pollution problem by co-located non-RT servers and the latter priority inversion in a guest kernel. The proposed method addresses them by socket outsourcing, in which a guest kernel delegates network processing to the host kernel. The proposed method achieved consistent latency. Compared to the threaded interrupt handling method, the proposed method reduced the standard deviation (SD) of the latencies of a simple RT server by a factor of 6, achieving 5.6% higher throughput and 32% lower CPU utilization. Compared to the exclusive CPU method, the proposed method reduced the SD by a factor of 2 and prevented underutilization of the exclusive CPU. The proposed method was more scalable in terms of the number of RT VMs. A four-CPU host was able to execute 40 RT VMs using the proposed method while maintaining the throughputs of non-RT servers.

INDEX TERMS KVM, Linux, priority inversion, real-time latency, socket outsourcing.

I. INTRODUCTION

Web-facing applications with stringent quality-of-service requirements (service level agreements, e.g., near-zero latency), are being increasingly deployed in computing clouds owing to the scalability of n-tier architectures and virtual machine (VM) environments. However, the longer execution and data flow paths in such environments also introduce significant challenges to real-time (RT) applications (predictable latency requirements) owing to the increased variance in network latency. Although advances in network hardware (e.g., 100-Gb optical fiber and 5G wireless connections) have reduced data transmission time, there is an increasing impact on the execution time variance owing to the lengthened (and increasingly complex) network protocol stacks in virtualized environments.

End-to-end execution time variance in n-tier systems often follows a negative exponential distribution with a long tail, e.g., with the dominant majority of requests returning within a few milliseconds, but with a small percentage taking a much longer time (in the order of seconds). We acknowledge that there are many speculated sources of long-tail latency [1] and confirmed sources, such as the queueing effects caused by millibottlenecks on resources that include CPUs, memory, and I/O [2]–[4]. In this paper, we focus on the variances introduced by the network protocol stacks, which are considerable in commodity hosted VM environments such as Linux KVM.

When we run RT servers and non-RT servers together in a hosted VM environment, we give higher priorities to the threads of the RT servers. Nonetheless, non-RT servers can interfere with RT servers and cause variances to the

latter's response times. This means that there exist priority inversion problems in the message processing path from interrupts to the RT servers. It is not trivial to find such priority inversion problems in complex network stacks of hosted VM environments.

We performed experiments and analyzed the message processing path of RT and non-RT servers in vanilla (non-RT) Linux and two production RT methods for Linux. We found three major sources of variances. Two are priority inversion problems during interrupt handling in the host operating system (OS) and one is cache pollution by co-located non-RT servers (Section II). Vanilla Linux has a priority inversion problem in interrupt handling and executes interrupt handlers first, prior to any high-priority processes including threads of virtual machines (Section II-A). One production method for Linux uses the PREEMPT_RT patch [5], which executes interrupt handlers by threads with their own priorities and addresses this priority inversion problem (Section II-B). We call this method *the threaded interrupt handling method*. Whereas threaded interrupt handling eliminates the first priority inversion problem, the second one in softirq handling of the host kernel remains (Section II-C). The second production method allocates an exclusive CPU to a group of host RT threads. We call this method *the exclusive CPU method* (Section II-D). Although the exclusive CPU method removes the second priority inversion problem, it has two disadvantages: low utilization of exclusive RT CPUs and low throughput of co-located non-RT servers. Furthermore, this method has the same cache pollution problem that the threaded interrupt handling method has (Section II-E).

To solve these problems, we propose a new approach to an RT network stack in a Linux KVM-based virtual machine environment. We call our approach the "socket outsourcing with partitioned RT softirq handling" method or *the outsourcing method* for short. First, we address the priority inversion problem in the interrupt-first host kernel using the PREEMPT_RT patch (Section II-B). Next, we address the priority inversion problem in the host's softirq handling by dividing softirq handling into RT and non-RT (Section III-A). Finally, we mitigate the cache pollution problem and prevent the priority inversion problem in a guest's softirq handling by extending socket outsourcing [6] (Section III-C).

The main contribution of the paper is an experimental demonstration showing that RT network stacks are achievable by using the outsourcing method, whose implementation requires small modifications to Linux KVM, a commodity OS-based VM environment. Compared to the threaded interrupt handling method, the outsourcing method reduced the standard deviation of the latencies of a simple RT server by a factor of 6. At the same time, the outsourcing method improved the non-RT throughput by up to 5.6% with 32% lower CPU utilization. Compared to the exclusive CPU method, the outsourcing method reduced the standard deviation by a factor of 2 and avoided low utilization of the exclusive RT CPU (Section IV-B). Moreover, the outsourcing method was effective for running two time-sensitive applications: a Voice-over-IP (VoIP) server and a key-value store server (Section IV-G).

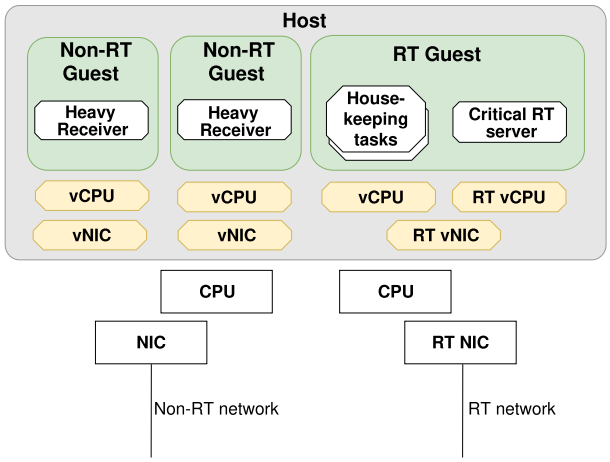

FIGURE 1. Running an RT server and co-located non-RT servers in a virtual machine environment.

The outsourcing method was more scalable in terms of the number of RT VMs. Our experimental results showed that a four-CPU host was able to execute 40 RT VMs using the outsourcing method while maintaining the throughputs of non-RT servers.

This paper builds on of our research [7] and incorporates the following updates:

1) Evaluation of scalability in the number of RT VMs.
2) Detailed analysis of message processing paths of an RT server. This clarifies sources of latency and latency variances.
3) Detailed analyses of message processing paths of a non-RT server. This clarifies the interferences caused by co-located non-RT servers.
4) Evaluation of the extended socket outsourcing by comparing with conventional socket outsourcing.

In addition, we have fixed some problems in priority parameters and performed all experiments again.

II. FINDING CAUSES OF LATENCY VARIANCES IN VANILLA LINUX AND TWO PRODUCTION RT METHODS

In this section, we illustrate the causes of latency variances in vanilla Linux and two representative production RT methods. We use a typical virtual machine environment that hosts two types of servers (Figure 1):

- *Critical RT server*. Receives requests from clients occasionally and sends response messages to the clients. It requires short and consistent response times.
- *Heavy Receiver*. Receives messages persistently from clients at the maximum speed and stresses the receiver-side of the network stack. However, it does not send response messages. A Heavy Receiver requires high throughput.

In this figure, we run one of these servers in an individual VM. Each VM has one or two vCPUs, which are implemented

by a host thread called *vCPU thread*. The VM of each Heavy Receiver has a single vCPU thread with a normal priority. The VM of the Critical RT server has two vCPUs, as in [8]. One is a non-RT vCPU thread with a normal priority and executes system tasks (e.g., housekeeping tasks) in the guest. The other is an RT vCPU thread with a high RT priority,[1] and executes the Critical RT server in the guest. We assume that a Critical RT server requires a small amount of the CPU resources and the Heavy Receivers use the rest of the CPU resources.

In Figure 1, the host is connected to the following two networks:

- *The RT network*. The bandwidth and delay and are guaranteed as in [9]–[12]. The network interface card (NIC) to this network is labeled as RT NIC.
- *The non-RT network*. This is a best-effort network.

In Figure 1, each VM has a vNIC thread that executes a backend network driver of the VM. The host directs messages from the RT network to the Critical RT server and those from the non-RT network to either of the Heavy Receivers. The vNIC thread of a Heavy Receiver runs with a normal priority, whereas that of the Critical RT server runs with a high RT priority.

A. THE PRIORITY INVERSION PROBLEM IN THE INTERRUPT-FIRST HOST KERNEL OF VANILLA LINUX

In this subsection, we describe the priority inversion problem in the interrupt handling of vanilla Linux. Figure 2a illustrates the interrupt handling of the RT and non-RT NIC in vanilla Linux. Each NIC has two interrupt handlers: the *hard Interrupt Request (IRQ) handler* and the *softirq handler*. The former executes the essential interrupt tasks while interrupts from the device are disabled. In contrast, the latter executes the rest of the interrupt tasks, including heavy TCP and bridge processing, typically after enabling interrupts. Drivers of high-performance NICs can use the polling mode [13]. The softirq handler of such a driver drains packets from a NIC while interrupts from the NIC are disabled. After that, the driver enables interrupts and hands control over to the upper layers.

A device driver can create multiple hard IRQ and softirq handlers for receiving multiple messages in parallel. For example, the device driver of the Intel $\times$520 NIC creates multiple hard IRQ and softirq handlers (up to 64) for multiple CPU cores [14]. In Figure 2a, each device driver in the host OS has two hard IRQ and two softirq instances for the two physical CPUs. On the other hand, each device driver in a guest OS is a paravirtual driver and creates a single hard IRQ and a softirq handler.

A NIC injects IRQs into arbitrary CPUs by default. In Figure 2a, when a CPU receives an IRQ from a NIC in the host OS, the CPU suspends the current running process and executes the hard IRQ handler that is bound to the CPU. Each CPU has its own instance of the softirq mechanism with per-CPU variables, and multiple NIC drivers share these instances. The CPU receives the IRQ from the NIC and executes both the hard IRQ and the softirq handler for cache affinity. This is implemented through the *poll_list*, which is a per-CPU variable in the softirq mechanism and contains softirq handlers with interrupt tasks. The hard IRQ handler of a NIC inserts the RT softirq handler into the poll_list of the current CPU. After completing the hard IRQ handler, the CPU enters its instance of the softirq mechanism. The CPU acquires the lock of the instance called *softirq_lock*, executes each pending softirq handler in the poll_list, and releases the softirq_lock. In a guest OS of Figure 2a, on the other hand, each VM has a single vNIC with a hard IRQ and a softirq handler. The vNIC of the Critical RT server injects virtual interrupts into the RT vCPU, and this vCPU executes the hard IRQ and softirq handlers in the RT guest OS.

This interrupt handling has a priority inversion problem. That is, the kernel of vanilla Linux executes interrupt handlers first, prior to user processes. In Figure 2a, for instance, while the host kernel is executing the RT vCPU thread with a high priority, the kernel can execute the hard IRQ and softirq handlers of a non-RT NIC. We address this priority inversion problem in this paper.

This interrupt handling mechanism increases the latency variance of the RT server. On the other hand, this mechanism has an advantage in that it can yield high CPU utilization and high throughput because all CPUs execute any vCPU and vNIC threads.

In a typical hosted VM environment, a network message is handled by both the host kernel and a guest kernel. This often causes extra message copying. In Figure 2a, a vNIC thread of the host kernel and a guest kernel perform message copying. Section II-E will discuss this copying problem.

B. THREADED INTERRUPT HANDLING METHOD

To address the priority inversion problem discussed in Section II-A, a production method uses the PREEMPT_RT patch [5]. We call this method the *threaded interrupt handling method*. Applying the PREEMPT_RT patch transforms the kernel into a more preemptible one because of the following characteristics:

- Interrupt handlers are executed by threads (interrupt handler threads). When a CPU receives an interrupt, the CPU wakes an interrupt thread, which executes the corresponding hard IRQ and softirq handlers.
- The patch translates spin locks into mutexes that implement a priority inheritance protocol.

Figure 2b illustrates interrupt handling in the threaded interrupt handling method. Because this host has two physical CPUs, the driver of each NIC creates two interrupt handler threads for the two CPUs. Each interrupt handler thread is bound to one CPU.

This method eliminates the priority inversion problem in Section II-A as follows. Each interrupt handler thread executes the hard IRQ and softirq handlers with its own priority.

[1] The Linux kernel executes the processes with a high RT priority in preference to the processes with a normal priority. The processes with a normal priority are scheduled by the Completely Fair Scheduler.

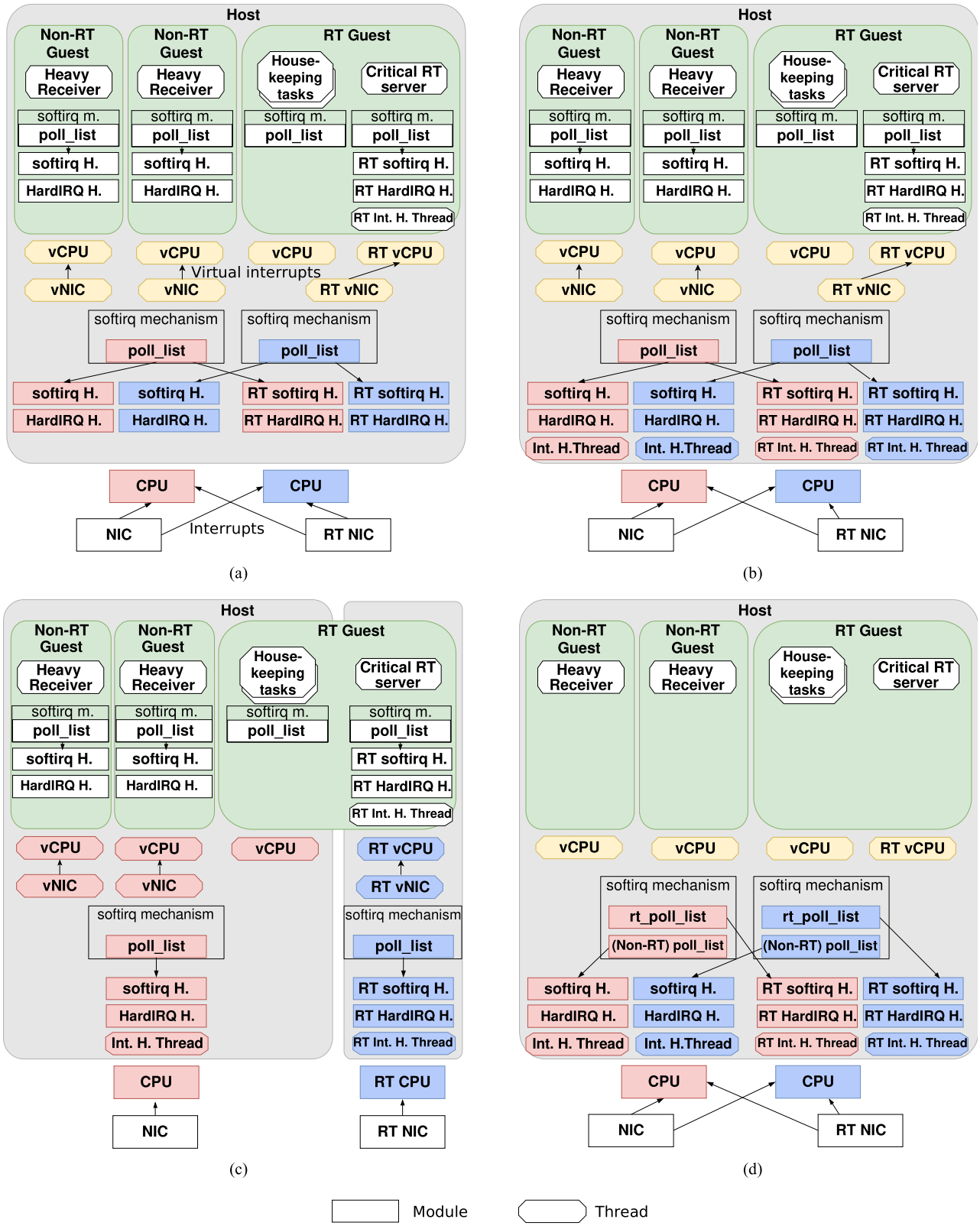

FIGURE 2. Interrupt handling by vanilla Linux and the RT methods. (a) Vanilla Linux. (b) Threaded interrupt handling. (c) Exclusive CPU. (d) Outsourcing.

In Figure 2b, for example, the interrupt handler thread of the non-RT NIC has a normal priority and does not preempt the threads of the RT VM. In addition, because all CPUs execute any vCPU and vNIC threads as in vanilla Linux, this method can also produce high CPU utilization and high throughput, as vanilla Linux does.

C. THE PRIORITY INVERSION IN THE SOFTIRQ MECHANISM OF LINUX

The threaded interrupt handling method eliminates the priority inversion problem discussed in Section II-A. However, another type of priority inversion problem remains in the softirq handling of Linux. We have found this through analyzing the message processing path from interrupts to the RT servers using KernelShark [15].[2]

Figure 3 shows a trace of interrupt handling using the threaded interrupt handling method. As shown in this figure, while the CPU was executing a user process, a non-RT NIC injected an IRQ to the CPU. Then, the CPU woke the interrupt handler thread of the non-RT NIC driver, and this thread executed the non-RT hard IRQ handler. The interrupt handler thread of the non-RT driver placed the non-RT softirq handler into the CPU's poll_list.

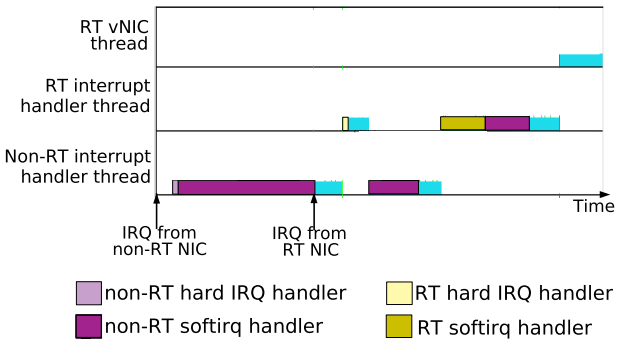

FIGURE 3. Priority inversion in the softirq handling in the host OS.

Next, the interrupt handler thread of the non-RT driver entered the softirq mechanism of the CPU. The thread of the non-RT driver acquired the softirq_lock of the CPU and executed the non-RT softirq handler.

In Figure 3, while the CPU was executing the non-RT softirq handler, an RT NIC injected an IRQ to the CPU. The CPU preempted the interrupt handler of the non-RT driver and woke the interrupt handler thread of the RT NIC driver. Because the interrupt handler thread of the RT NIC driver had a higher priority than that of the non-RT driver, the CPU executed the former thread. This thread executed the RT hard IRQ handler, which inserted the RT softirq handler into the poll_list of the CPU.

Next, the interrupt handler thread of the RT NIC driver entered the softirq mechanism of the CPU. This thread attempted to acquire the softirq_lock. However, it was already locked by the non-RT interrupt handler thread. Therefore, the CPU suspended the interrupt handler thread of the RT NIC driver and executed the interrupt handler thread of the non-RT NIC driver. This thread then executed the non-RT softirq handler. At this time, this thread executed the non-RT softirq handler with a high priority based on the priority inheritance protocol. The interrupt handler thread of the RT NIC driver with a higher priority had to wait until the non-RT thread with a lower priority finished. This indicates that there was a priority inversion.

Next, in Figure 3, the non-RT softirq handler exceeded the execution quota limit in the number of iterations. Therefore, the interrupt handler thread of the non-RT driver placed the non-RT softirq handler at the end of the poll_list, released the softirq_lock, and went to sleep. Because the softirq_lock was released, the interrupt handler thread of the RT NIC driver became executable and the CPU executed it. The RT NIC driver thread acquired the softirq_lock and executed the RT softirq handler. This handler processed network messages from the RT NIC, placed them in a queue, and then woke the RT vNIC thread. Next, the RT interrupt handler thread obtained the non-RT softirq handler from the poll_list and executed it with high priority. This means that there was a virtual priority inversion because the RT vNIC thread had to wait. Finally, the interrupt handler thread finished the non-RT softirq handler, released the softirq_lock, and yielded the CPU to the vNIC thread.

D. EXCLUSIVE CPU METHOD

Section II-C described the priority inversion in the softirq mechanism of Linux. A popular production technique eliminates this by allocating an exclusive CPU to a group of RT threads in commodity hosted environments [8], [16], [17]. In this paper, we call this technique *the exclusive CPU method*.

Figure 2b illustrates interrupt handling in the exclusive CPU method. In this method, we allocate an exclusive CPU (labeled as RT CPU) to a group of threads that execute the tasks of the Critical RT server. The driver of the RT NIC has a single interrupt thread, hard IRQ handler, and softirq handler. The RT NIC injects interrupts only to the RT CPU. The RT CPU executes this interrupt thread, the RT vNIC thread, and the RT vCPU in the host.

On the other hand, the driver of the non-RT NIC has its own set of a single interrupt thread, hard IRQ handler, and softirq handler. These are shared by the VMs of the two Heavy Receivers. The non-RT NIC injects interrupts only to the non-RT CPU. This means that interrupt handling of the non-RT NIC driver never disturbs that of the RT NIC driver. Therefore, there is no priority inversion problem as discussed in Section II-C in the exclusive CPU method.

Although this method can achieve a consistently low latency, it has a drawback. Because RT CPUs do not help in the execution of non-RT threads, this method yields lower CPU utilization.

E. CACHE POLLUTION BY CO-LOCATED NON-RT SERVERS

When we run RT servers and non-RT servers together in a virtual machine environment, as shown in Figure 1, we can control the allocation of CPU (cores) using priorities of threads and CPU isolation. On the other hand, it is not trivial to control the Last Level Cache (LLC) without recent advanced hardware support, such as Intel's Cache Allocation Technology (CAT) [18] and ARM's Cache Lockdown [19].

[2] Unlike a hardware monitor in Section IV-A or lightweight probes in Section IV-D, these results included large problem effects.

For example, Intel Core i7 does not have such capability. In such an environment, co-located non-RT servers pollute the LLC and interfere with RT servers. In Figure 2b, for example, the Heavy Receivers are receiving a large number of messages persistently from clients. In a regular hosted VM environment, this causes same-message copying in both the vNIC threads and guest OSs. This duplicated copying pollutes the LLC, which causes latency variance in the Critical RT server.

It is worth mentioning that not only the threaded interrupt handling method but also the exclusive CPU method has this problem. We cannot dedicate a partition of LLC to RT servers without advanced hardware support. In the next section, we propose a software-based method for mitigating this LLC pollution problem. This method works in commodity hosted VM environments without requiring such advanced hardware support.

III. OUTSOURCING METHOD

In Section II-B and Section II-D, we have described two production RT methods, namely, the threaded interrupt handling method and the exclusive CPU method. In this section, we describe our proposed method, the ''socket outsourcing with partitioned RT softirq handling'' method or *the outsourcing method* for short.

Figure 2d shows the interrupt handling of the outsourcing method. Similar to the threaded interrupt handling method, this method avoids the priority inversion problem in the interrupt-first host kernel, described in Section II-A, by using the PREEMPT_RT patch and assigning high priorities to RT threads. Second, this method avoids the priority inversion problem in the host's softirq handling, described in Section II-C, by dividing softirq handling into RT and non-RT types (Section III-A). Lastly, this method mitigates the LLC pollution problem described in Section II-E and avoids the priority inversion problem in a guest's softirq handling by extending conventional socket outsourcing [6] (Section III-C).

A. DIVIDING SOFTIRQ HANDLING INTO RT AND NON-RT TYPES

The outsourcing method divides the poll_list of the softirq mechanism into the following two types (Figure 2d).

- **(non-RT) poll_list:** The poll_list for non-RT softirq handlers.
- ***rt_poll_list:*** The poll_list for RT softirq handlers.

Similarly, we divide the softirq_lock into two locks: (non-RT) softirq_lock and *rt_softirq_lock*.

Figure 4 shows a KernelShark trace of interrupt handling using the outsourcing method. As in Figure 3 in Section II-C, this CPU received two IRQs. The first was from the non-RT NIC and the second was from the RT NIC.

First, the CPU performed typical functions until the IRQ from the RT NIC arrived. The CPU woke the interrupt handler thread of the non-RT NIC driver, and this thread executed the non-RT hard IRQ handler. The interrupt handler thread of the non-RT driver inserted the non-RT softirq handler into the CPU's *non-RT* poll_list.

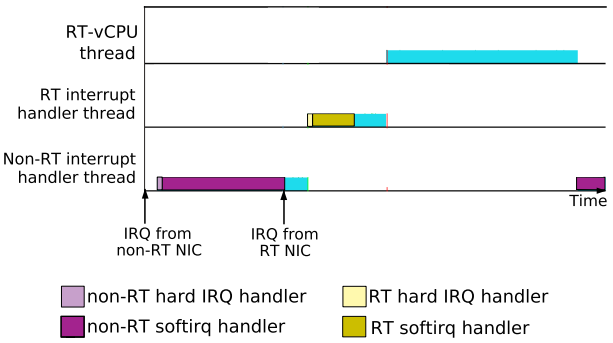

FIGURE 4. Separation of RT softirq handling from non-RT softirq handling.

Next, the interrupt handler thread of the non-RT driver entered the softirq mechanism of the CPU. The thread of the non-RT driver acquired the *non-RT* softirq_lock of the CPU and executed the non-RT softirq handler.

In Figure 4, while the CPU was executing the non-RT softirq handler, an RT NIC injected an IRQ into the CPU. The CPU preempted the interrupt handler of the non-RT driver and woke the interrupt handler thread of the RT NIC driver. Because the interrupt handler thread of the RT NIC driver had a higher priority than that of the non-RT driver, the CPU executed the former thread. This thread executed the RT hard IRQ handler, which inserted the RT softirq handler into the rt_poll_list instead of the non-RT poll_list.

Next, the interrupt handler thread of the RT NIC driver entered the softirq mechanism of the CPU. Unlike in Figure 3, this thread acquired the rt_softirq_lock instead of the softirq_lock and executed the RT softirq handler. In contrast to the threaded interrupt handling method in Figure 3, this thread executed the RT softirq handler but did not execute the non-RT softirq handler because the rt_poll_list only contained the RT softirq handler. This handler processed network messages from the RT NIC, placed them in a queue, and then woke the RT vNIC thread. Finally, the interrupt handler thread finished the softirq handler, released the rt_softirq_lock, and the CPU became available to the RT vCPU thread. In contrast to Figure 3, Figure 4 indicates no priority inversion in the softirq handling.

We have implemented this softirq mechanism in Linux. First, we have added the rt_poll_list and rt_softirq_lock to softnet_data, which is a per-CPU variable in the softirq mechanism. Next, we have copied the base functions of poll_list and softirq_lock, modified them slightly, and obtained those for RT. For example, we have copied the `net_rx_action()` function that processes the softirq handlers in the poll_list. We have modified the function to use the rt_poll_list instead of poll_list and obtained `rt_net_rx_action()`. Finally, we have changed the function `napi_schedule_irqoff()`, which is used by hard IRQ handlers to place corresponding softirq handlers in the poll_list.

We have modified this function to use either the rt_poll_list or non-RT poll_list according to the priority of the calling interrupt handler thread. Because we did not change the interface of `napi_schedule_irqoff()`, we can reuse the existing device drivers of NICs without any changes.[3] We have added the sysctl parameter `net.core.rtnet_prio` for choosing either the rt_poll_list or the non-RT poll_list. For example, if a system administrator sets the parameter with the command `sysctl -w net.core.rtnet_prio=47`, interrupt handler threads with a priority higher or equal to 47 use the rt_poll_list in `napi_schedule_irqoff()`.

These modifications required changing 150 lines of code and they are independent of the virtual machine monitor, Linux KVM. They can also reduce the latency of RT servers in non-virtualized environments and container-based virtual environments.

B. CONVENTIONAL SOCKET OUTSOURCING

To overcome the cache pollution problem and the priority inversion problem in a guest, we extend socket outsourcing [6]. Socket outsourcing is a technique used to realize fast networking, similar to paravirtualization. However, it differs in that socket outsourcing delegates high-level operations from a guest kernel to the host kernel while paravirtualization performs driver-level operations. For example, when a guest user process issues the `recvfrom()` system call, the guest kernel delegates its processing to the host. The network stack of the host receives messages from a NIC and passes them to the guest process directly.

The implementation of socket outsourcing uses Virtual Machine Remote Procedure Call (VMRPC) [6] as a communication mechanism between a guest kernel and the host kernel in a hosted VM environment. In VMRPC, a client in a guest kernel sends request messages to a server in the host kernel, and the server in the host kernel sends back the response messages to the client in the guest kernel. These request and response messages of VMRPC are transferred using the shared memory between guest and host.

Similar to execution of system calls and regular Remote Procedure Calls (RPCs) in distributed systems, VMRPC blocks clients. This means that a straightforward invocation stops the entire guest process of a client until the server of the host sends back a response message. To address this problem, conventional socket outsourcing makes use of virtual interrupts. For example, when a client in a guest performs a VMRPC to the procedure `recvfrom()` in the host server, the procedure should not block. Even though there is no message, this procedure returns immediately. The guest client puts the current process into sleep mode and changes the context to another process. When a message arrives at the host, the host's server sends a virtual interrupt into the guest. The interrupt handler of the guest wakes the receiving process and the process calls the procedure `recvfrom()` in the host again.

The current production RT methods, i.e., the threaded interrupt handling method and the exclusive CPU method, perform message copying two times. One occurs from the host kernel to a guest kernel, and the other occurs from the guest kernel to a guest user process. In contrast, socket outsourcing requires message copying only once, from the host kernel to a guest user process. When a guest process invokes receive and send procedures (e.g. `recvfrom()` and `sendto()`), the host translates the address of the buffer in the guest user process to that in the host kernel. The host performs these socket procedures in the same way as for regular user processes.

We have realized socket outsourcing in various guest operating systems, including Windows and NetBSD [6]. We used socket outsourcing for improving security through natural diversity, as well. We can run Internet Information Services (IIS) of a Windows guest using the network stack of the Linux host. In this case, we can defend against attacking the network stack of Windows.

C. RT SOCKET OUTSOURCING

Conventional socket outsourcing can face the priority inversion problem in the softirq mechanism as described in Section II-C because it makes use of virtual interrupts as described in Section III-B. We avoid this problem by removing interrupt handling from a guest OS for receiving RT messages. We call this new mechanism *RT socket outsourcing*.

We implement RT socket outsourcing by extending *the idle process*. In Linux, the idle process is a special kernel thread, and the scheduler executes the idle process when there is no runnable process in the ready queue. The idle process usually executes the halt instruction, and this stops the physical CPU if this is executed in the host. The CPU will resume when the CPU receives an interrupt. In RT socket outsourcing, the idle process in the guest OS also executes the halt instruction and this places the vCPU thread into sleep mode in the host.

When the host receives a new message, the host vCPU thread wakes. This vCPU thread enters the virtual machine and executes the next instruction of the halt instruction in the idle process. The extended idle process in RT socket outsourcing reads an event queue and the states of the sockets in the shared memory. Next, the extended idle process makes the receiving process runnable and returns to the scheduler. The scheduler finds the receiving process immediately without interrupt handling and executes the process.

This mechanism has an advantage in that the receiving process does not disturb a running RT server. In other words, the guest kernel handles messages for an RT server in a first-in-first-out (FIFO) manner. When the RT server is processing a previous request message and a new message arrives, the guest kernel does not handle the new message immediately. The guest kernel handles it when the RT server completes processing the previous message and issues a system

[3] In Section IV, we used the device driver of the Intel ×520 NIC. We did not change any code of the device driver.

call to receive a new message or if the guest kernel becomes idle.

We have decided to modify the idle process because of the following reasons. The idle process is a safe point that watches not only the events for sockets but also the regular interrupts. We did not have to modify the existing interrupt handling process on a guest OS. For example, a guest OS can handle timer interrupts and inter-processor interrupts in the idle process. Section IV-C compares this RT socket outsourcing with conventional interrupt-based socket outsourcing.

We implemented RT socket outsourcing mainly as loadable kernel modules. The kernel module for a guest overrides the functions of the socket layer. Overridden functions send requests to the server in the host using VMRPCs. The kernel module for the host is the server that handles any requests from the guest. We also extended the idle process in the guest. This idle process calls a function that examines the event queue and the states of the sockets in the shared memory.

IV. EXPERIMENTAL EVALUATION

In Section III, we have described our proposed outsourcing method. In this section, we evaluate the outsourcing method by comparing it with the two production RT methods. First, we performed experiments using a simple RT server to show that the outsourcing method was able to reduce the latency and latency variances by eliminating the causes of the problems discussed in Section II. We also analyzed the execution paths of the outsourcing method and other production RT methods. Next, we ran a Voice-over-IP (VoIP) server and a key-value store server as an RT server. Finally, we evaluated scalability of the outsourcing method in the number of RT VMs.

A. EXPERIMENTAL SETUP FOR RUNNING A SIMPLE RT SERVER

We have performed experiments using a simple RT server in the experimental environment shown in Figure 5. First, we ran netperf [20] as the Critical RT server. We have slightly modified the client of netperf, which sent requests at random inter-arrival times ranging from 1 to 10 ms using UDP. We used iperf [21] in server mode as a Heavy Receiver. A Heavy Sender was the client of iperf and it transmitted messages persistently using TCP at the maximum speed.

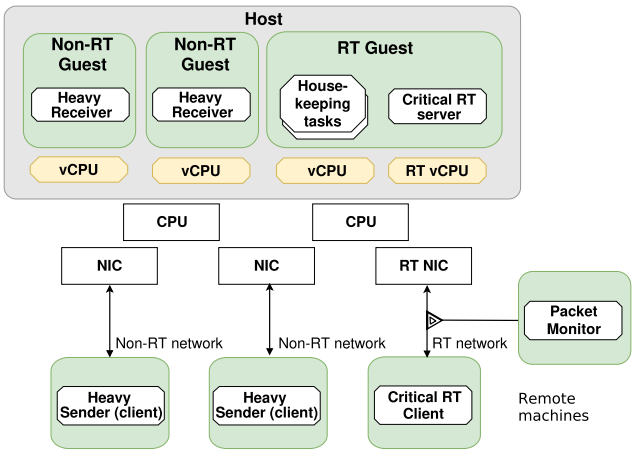

FIGURE 5. The experimental environment.

We emphasize that varying inter-arrival times caused a similar impact to the LLC as varying the non-RT workload. If the heavy sender sends messages at a fixed rate, its impact to the LLC is unchanged. Such a fixed impact can lead to steady results. We should avoid this (by using random intervals) because we measure the latency variance using these RT methods. We could vary the non-RT workload by changing the client of iperf. However, this was not easy. Therefore, we decided to mimic varying the non-RT workload throughput by varying inter-arrival times of the RT client, which was easy to do. When a client of the Critical RT server sent request messages at a shorter inter-arrival time, the LLC retained more contents of the Critical RT server. This means that the load of the non-RT Heavy Receiver was lower. When the client sent messages at a longer inter-arrival time, the LLC retained fewer contents of the Critical RT server. This means that the load of the non-RT Heavy Receiver was higher.

As shown in Figure 5, the host of the VMs was connected with three networks. One was an RT network and the other two were non-RT networks. All the networks consisted of 10GBASE-LR Ethernets over optical fibers. We used Intel ×520 Ethernet converged network adapters as the NICs. We connected the VM host to two non-RT network links to use up the CPU resources of the host. We performed a preparatory experiment and found that using a single link was not sufficient to use up the CPU resources because the bottleneck was the network link. The maximum transfer unit (MTU) of these networks was set to the default value, 1500 bytes.

We measured the latency, e.g., the response times of the Critical RT server at the RT network, with a hardware monitor (an Endace DAG 10×2-S card [22]). We chose to use the hardware monitor because it had no probe effect. The RT network in Figure 5 consisted of two optical links. Each link had an optical splitter that divides signals into two destinations. One destination was a network peer and the other destination was the hardware monitor. The hardware monitor took both the request and the response packets, timestamped them at a resolution of 4 ns, and saved them into a file. Note that the obtained results included delays in the NIC of the server, but did not include any delays on the client side.

In the experiments, we used the physical machine in Table 1. The CPUs were Intel Core i7. We activated two of four cores of the VM host to measure response times for a single RT server. In the exclusive CPU method, we allocated a CPU as the non-RT CPU and another CPU as the RT CPU. This RT CPU ran a group of RT threads as discussed in Section II-D. In other methods including vanilla Linux, both CPUs ran any threads. We activated all the cores of the other machines. The OSs running on the physical machine were Linux 4.1 except for the packet monitor. The machine for the

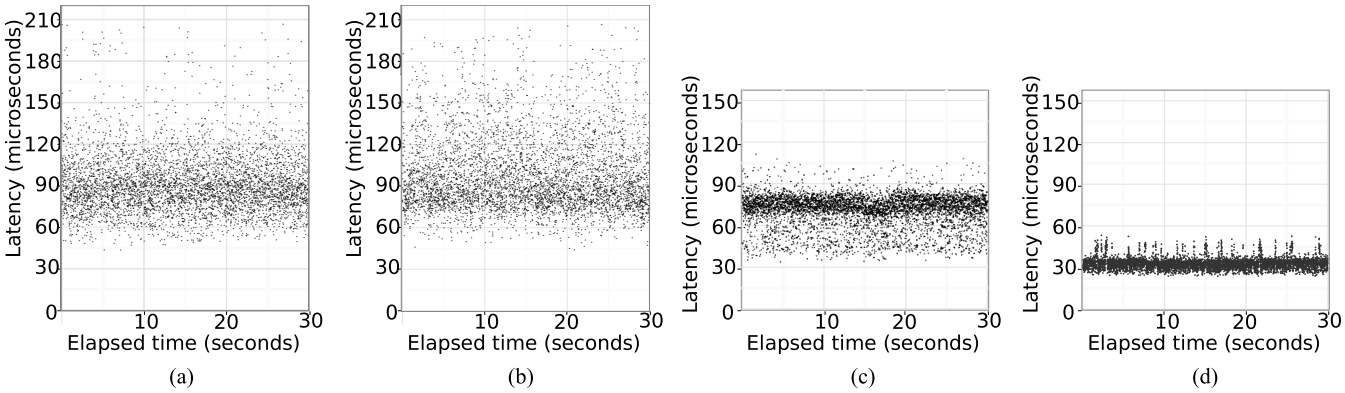

FIGURE 6. Distribution of the Critical RT server response times. (a) Vanilla Linux. (b) Threaded interrupt handling. (c) Exclusive CPU. (d) Outsourcing.

TABLE 1. Specifications of the machines and their active cores in the experiments.

Machine	CPU / Cache (MB)	Active cores	OS
VM host	Intel Core i7-6700K / 8	2	Linux 4.1
Critical RT client	Intel Core i7-6700K / 8	4	Linux 4.1
Heavy Sender (client) 1	Intel Core i7-3820 / 10	4	Linux 4.1
Heavy Sender (client) 2	Intel Core i7-3820 / 10	4	Linux 4.1
Packet monitor	Intel Core i7-3820 / 10	4	Linux 3.16

packet monitor ran on Linux 3.16. The version of all guest OSs was Linux 4.1.

To eliminate fluctuations in the hardware, we turned off the following hardware features: Hyper-Threading, TurboBoost, and C-States.[4] Further, we used the `CONFIG_NO_HZ_FULL` option in both the host and the guest kernel of the Critical RT server. This reduced the number of clock ticks in the physical CPU and vCPU.

In these experiments, we set high priorities to the RT threads and normal priorities to non-RT threads. Table 2 presents the scheduling policies and priorities of these threads. The threads with the FIFO scheduling policy have higher priorities than threads with the normal scheduling policy. Within the FIFO scheduling policy, a larger priority value indicates a higher priority. As described in Section III-A, we set `sysctl -w net.core.rtnet_prio=47` and made the RT IRQ handler use the rt_poll_list.

B. EXPERIMENTAL RESULTS USING A SIMPLE RT SERVER

We ran the simple RT server as in Section IV-A for 30 s using the following methods:

[4]C-states are CPU modes for saving power. C-state transitions degrade the performance of RT servers. We turned off C-states in the BIOS and in the Linux kernel using the parameters `intel_idle.max_cstate=0` and `idle=poll`.

TABLE 2. Scheduling policy and priority of the threads in the host OS.

Threads	Scheduling policy and priority
RT interrupt handler thread	FIFO(50)
RT vNIC thread	FIFO(48)
RT vCPU thread	FIFO(47)
Non-RT interrupt handler threads	Normal
Non-RT vNIC threads	Normal
Non-RT vCPU thread	Normal

- (non-RT) Vanilla Linux.
- The threaded interrupt handling method.
- The exclusive CPU method.
- The outsourcing method.

We obtained the latencies, the response times of the RT server shown in Figure 6, with the hardware monitor as described in Section IV-A. Table 3 summarizes the statistical values (the mean, 99th percentile, and standard deviation (SD)). At the same time, we measured the total throughputs of the Heavy Receivers and the CPU utilization of the VM host. These results are shown in Figure 7 and Figure 8, respectively.

TABLE 3. Statistical values of the Critical RT server response times (microseconds).

Method	Mean	99th percentile	Standard deviation
(non-RT) Vanilla Linux	92.5	225.0	29.2
Threaded interrupt handling	100.8	202.9	29.0
Exclusive CPU	70.5	96.0	11.8
Outsourcing	32.9	46.8	4.3

Vanilla Linux and the threaded interrupt handling method had high latency variance, as shown in Figure 6a and Figure 6b. In terms of the 99th percentile, the threaded interrupt handling method was better than vanilla Linux, as shown in Table 3. The total throughputs of the Heavy Receivers were high, as shown in Figure 7. They were 18.8 Gbps and 17.8 Gbps with the two 10-Gbps links. Figure 8 illustrates that vanilla Linux had spare CPU resources for running all the servers (the single Critical RT server and the two Heavy Receivers). The threaded interrupt handling method required

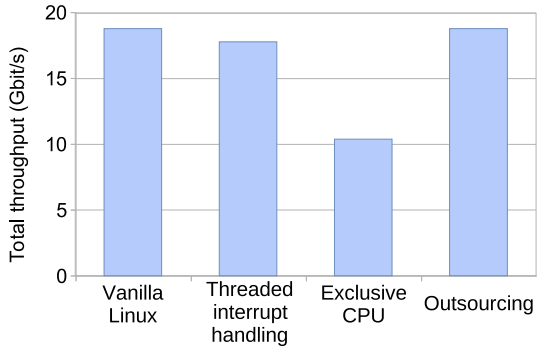

FIGURE 7. Total throughput of Heavy Receiver.

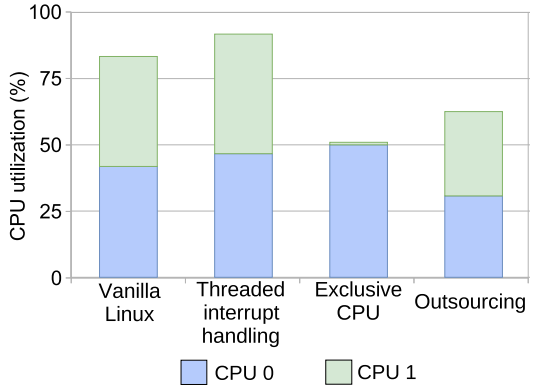

FIGURE 8. Achievable CPU utilization.

more CPU resources than vanilla Linux owing to the overhead produced by thread context switching.

The exclusive CPU method reduced the latency variance, as shown in Figure 6c compared with the previous two methods. The 99th percentile and standard deviation were around half of the previous two methods. However, the CPU utilization of the exclusive RT CPU was low, as shown in Figure 8. This bounded the total throughput to 10.4 Gbps.

The outsourcing method produced the lowest latency and latency variance among these methods, as shown in Figure 6d and Table 3. The mean, 99th percentile, and standard deviation were less than half of the exclusive CPU method. Furthermore, the outsourcing method produced the same high throughput of 18.8 Gbps with a lower CPU utilization than vanilla Linux.

In summary, compared to the threaded interrupt handling method, the outsourcing method reduced the standard deviation of the latencies of a simple RT server by a factor of 6 with 5.6% higher throughput and 32% lower CPU utilization. Compared to the exclusive CPU method, the outsourcing method had a lower standard deviation and a higher total throughput (by a factor of 2), and avoided low utilization of the RT CPU.

C. EFFECTS OF INDIVIDUAL TECHNIQUES

The outsourcing method comprises two techniques: adding the rt_poll_list (Section III-A) and RT socket outsourcing (Section III-C). We performed the same experiments as in Section IV-A by enabling one of two techniques at a time. Figure 9a shows the result using both techniques, and Figures 9b and 9c show the results using one of the two techniques without the other. When we enabled only one of the two techniques, we obtained larger variances of the response times than those obtained using both techniques.

In Section III-C, we described how to extend the idle process to eliminate virtual interrupt handling from a guest OS. We compared these two mechanisms using the same experiments as in Section IV-A. Figure 9a and Figure 9d show the results. Using the extended idle process (Figure 9a) showed better real-time characteristics than using virtual interrupts (Figure 9d).

D. PROCESSING PATH ANALYSIS WITH LIGHTWEIGHT PROBES

We analyzed the processing path of request messages from the RT NIC to the Critical RT server and their corresponding response messages from the Critical RT server to the RT NIC in detail using lightweight probes.

1) MEASURING LATENCIES WITH LIGHTWEIGHT PROBES

We have implemented lightweight probes for measuring latencies of components in hosted VM environments. Every lightweight probe is identified by a unique number. When a lightweight probe is executed, the probe takes a timestamp

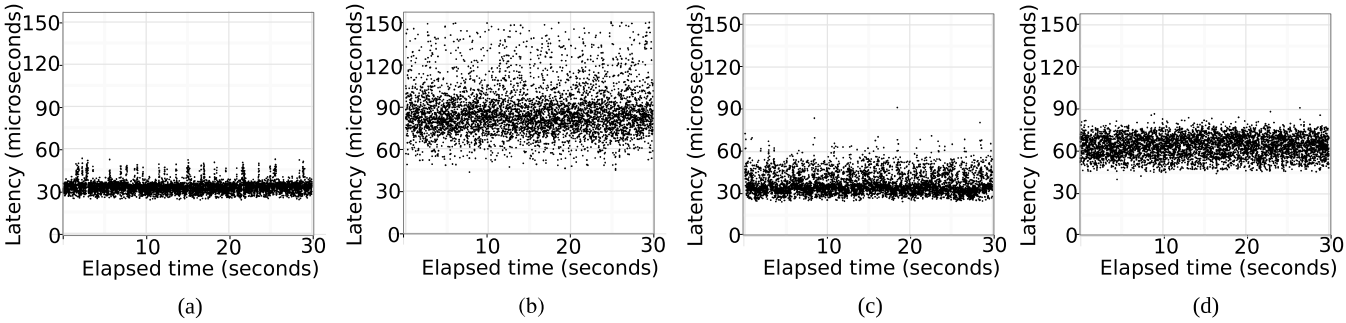

FIGURE 9. Distribution of the Critical RT server response times with the outsourcing method using individual techniques. (a) RT socket outsourcing (enabling two techniques and using the extended idle process). (b) Only adding the rt_poll_list. (c) Only using RT socket outsourcing. (d) Using virtual interrupts.

from the CPU instruction "read time-stamp counter (rdtsc)" and places the timestamp and its identification number into a buffer in the kernel memory. When the experiment finishes, the buffer is dumped into a file.

We measured the probe effect of lightweight probes using the hardware monitor (Endace DAG 10×2-S card). The probe effect was less than 0.5 μs when using four lightweight probes.

2) RESULTS OF PROCESSING PATH ANALYSIS

We divided the processing path into the following segments (Figure 10):

- **Host receive:** Host execution from the receipt of an IRQ to a VM entry.
- **Guest:** Guest execution from the VM entry to a VM exit when sending a message.
- **Host send:** Host execution from the VM exit to a message transmission to a NIC.

We inserted a lightweight probe at the beginning of each segment and at the end of the message transmission. Next, we repeated the experiments in Section IV-A.

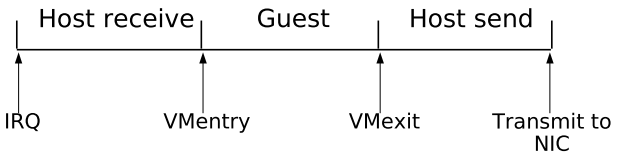

FIGURE 10. Division of the message processing path into three segments.

Figure 11 presents the results of these experiments. Figure 11a shows the results using vanilla Linux. By comparing Figures 11b, 11c, and 11d, we identified that most of the latency variances were located in the "host receive" segment and the "guest" segment. We found the two priority inversion problems in the "host receive" segment as described in Section II-A and Section II-C. These priority inversion problems were not present when using the exclusive CPU method, as described in Section II-D, and when using the outsourcing method, as described in Section III.

The threaded interrupt handling method and the exclusive CPU method had higher latency variances in the "guest" segment than the outsourcing method. This is because the execution of the non-RT Heavy Receivers polluted the LLC and removed the contents of the Critical RT servers.

E. CACHE POLLUTION IN THE RT METHODS

Using the same experiments as in Section IV-A, we analyzed the impact on the LLC. We obtained the numbers of cache references and misses using the perf command of Linux [23]. This command uses the hardware performance counters of cache references and cache misses. We calculated the cache miss ratio by dividing the number of cache misses by the number of references.

Figure 12 presents the results of these experiments. The cache pollution when using the outsourcing method was the lowest among the three methods because message processing had a smaller memory footprint. The cache pollution problem existed in the exclusive CPU method, as the Heavy Receivers also interfered with the execution of the Critical RT server because the RT CPU shared the LLC with the non-RT CPU.

F. MESSAGE PROCESSING PATHS OF NON-RT SERVER

The outsourcing method shortens the message processing path of co-located non-RT servers and reduces cache pollution by the non-RT servers. In this section, we analyze the message processing paths of a Heavy Receiver and confirm the reduction of cache pollution by using the same experiments as in Section IV-A.

Figure 13 compares the message processing paths using the threaded interrupt handling method and the outsourcing method. We obtained execution times by using the lightweight probes, as indicated in Section IV-D. In Figure 13, "User process" and "Socket processing" mean the execution of the Heavy Receiver and system call layer in a kernel. Figure 13b has two "Socket processing" executions. The upper one is the execution in the guest kernel and the lower one is that of the host kernel. In Figure 13, message copying is marked with diagonal lines.

Figure 13a illustrates the message processing path using the threaded interrupt handling method. In this method, message copying was performed two times, i.e., once between the host kernel and a guest kernel in the vNIC thread, and another from the guest kernel to the guest user process in the guest OS. By contrast, Figure 13b illustrates the message processing path using the outsourcing method. In this method, message copying was performed once, from the host kernel to the guest user process.

G. APPLICATION BENCHMARKS

In previous sections, we ran netperf as a Critical RT server and analyzed the fundamental features of RT methods. In this section, we ran two time-sensitive applications as a Critical RT server and compared these RT methods. The experimental environment and configurations were the same as those in Section IV-A.

1) A VOICE-OVER-IP (VOIP) SERVER

We ran a VoIP server as a Critical RT server and measured the forward delays of the VoIP server. The VoIP server was Kamailio [24], which exchanges messages based on the Session Initiation Protocol (SIP) [25]. We ran two SIPp [26] instances as communication peers of the Kamailio server in a remote machine. One instance acted as a user agent client (UAC) and the other acted as a user agent server (UAS). The VoIP server relayed messages between the UAC and the UAS.

Using the hardware monitor (Endace DAG 10×2-S card), we obtained the forward delays between the message that the VoIP server received and the message that the VoIP server sent during SIP calls. A single SIP call required forwarding

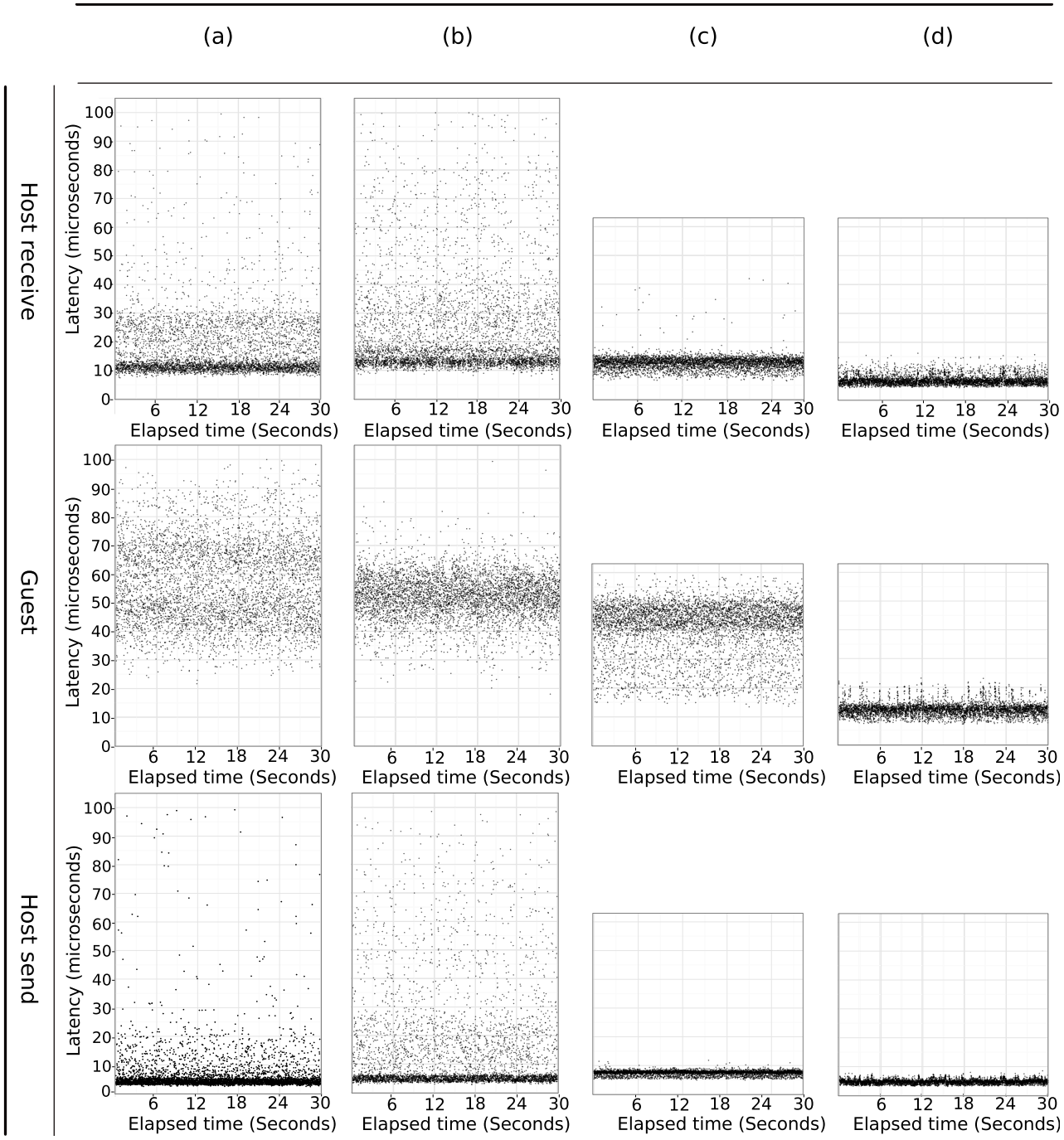

FIGURE 11. Latencies in three segments of the processing path of RT messages.

of the following six messages by the VoIP server.

1) An INVITE message from the UAC to UAS. (At this time, the server also sent a TRYING message to the UAC.)
2) A RINGING message from the UAS to the UAC.
3) An OK message from the UAS to the UAC.
4) An ACK message from the UAC to the UAS.
5) A BYE message from the UAC to the UAS.
6) An OK message from the UAS to the UAC.

Similar to using the modified netperf client in Section IV-A, we used a modified SIPp program. The modified SIPp program initiated SIP calls at random rates ranging from 17 to 167 calls per second. This means that the server forwarded 100 to 1000 messages per second. We ran the same Heavy Receivers employed in the previous experiments.

Figure 14 illustrates the percentiles (50^{th}, 99^{th}, and 99.9^{th}) of the forward delays. The outsourcing method had the lowest tail latencies among the RT methods. In the 99^{th} percentiles

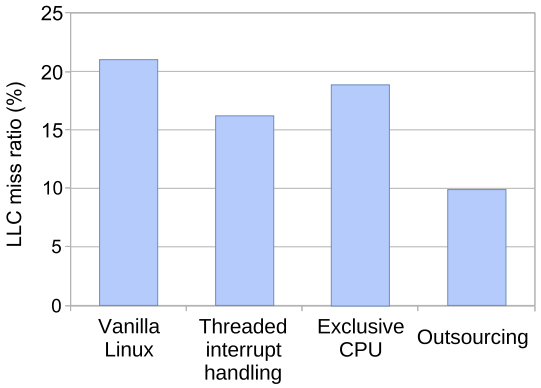

FIGURE 12. LLC miss ratio of the RT threads.

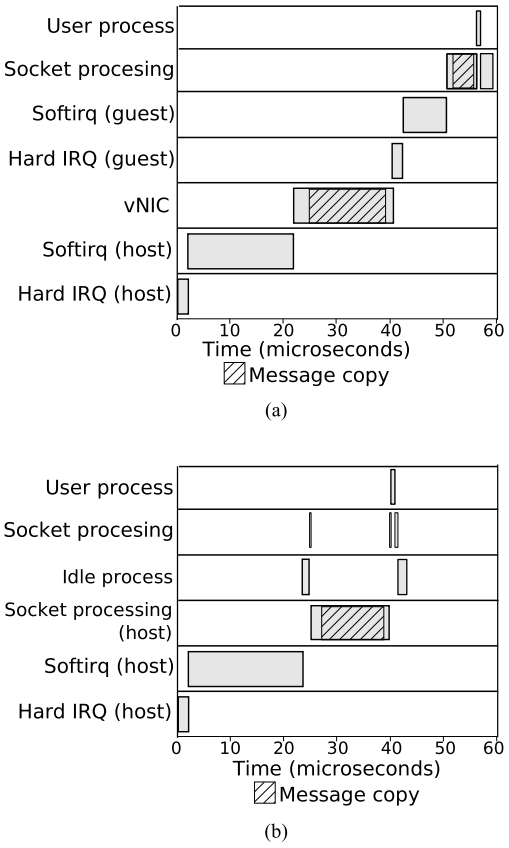

FIGURE 13. Message processing path of a non-RT Heavy Receiver. (a) Threaded interrupt handling. (b) Outsourcing.

results, for instance, the outsourcing method had 63% lower latency than the threaded interrupt handling method and 27% lower latency than the exclusive CPU method.

2) MEMCACHED

We ran Memcached [27] as a Critical RT server. Memcached is a distributed key-value store that is widely used for caching. The benchmark program was memaslap [28], which was executed in a remote machine.

Similar to using the modified netperf client in Section IV-A, we modified memaslap. The modified

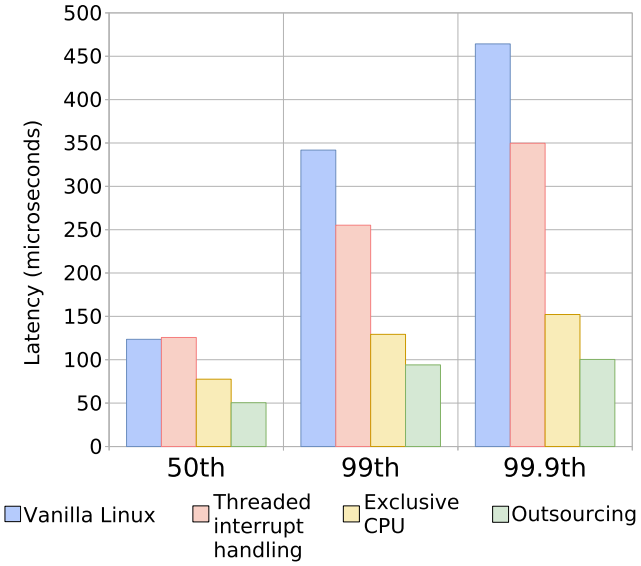

FIGURE 14. Forward delays of a Voice-over-IP server (Kamailio).

memaslap sent requests at random intervals ranging from 100 to 1000 requests per second. The size of a key was 64 bytes, and the size of the request value was 1024 bytes. Memaslap sent GET/SET requests at a ratio of 9:1. We measured the response times of the GET requests using the hardware monitor (Endace DAG 10×2-S card). We ran the same Heavy Receivers employed in the previous experiments.

Figure 15 illustrates the percentiles (50^{th}, 99^{th}, and 99.9^{th}) of the response times. The outsourcing method produced the best results among those RT methods. In the 99^{th} percentiles results, the outsourcing method exhibited 67% and 46% lower latency than the threaded interrupt handling method and the exclusive CPU method, respectively.

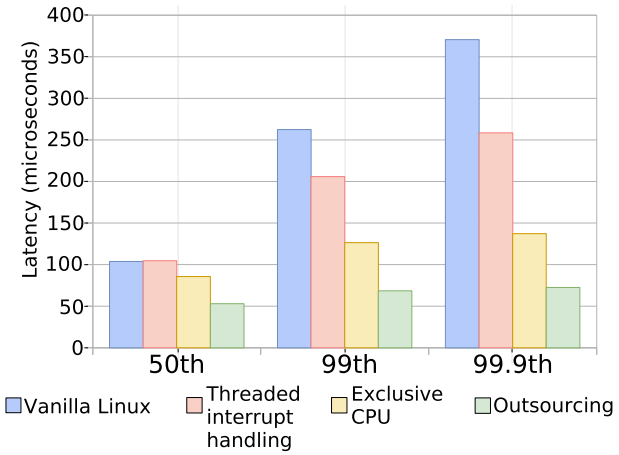

FIGURE 15. Response times of Memcached.

H. SCALABILITY OF RT VIRTUAL MACHINES

In Sections IV-B to IV-G, we fixed the number of RT VMs to one and we compared the latencies and the latency variances

of the RT methods. In this section, we increase the number of RT VMs and evaluate the scalability of these methods.

In this section, we performed experiments using the same configuration as in the previous sections except that we set the number of active CPU cores to four. We ran a Heavy Receiver in a single non-RT VM, and we ran two non-RT VMs as in the previous sections. We ran a Critical RT server in a single RT VM and increased the number of RT VMs up to 100 for the threaded interrupt handling method and the outsourcing method, and up to three for the exclusive CPU method. For the exclusive CPU method, we assigned one CPU as a non-RT CPU and the remaining CPUs as RT CPUs. We set high priorities to the RT threads and normal priorities to non-RT threads, as listed in Table 2. We ran the same number of clients in a remote machine as in the VMs. We measured the response times of the Critical RT servers using the hardware monitor (Endace DAG 10×2-S card).

Figures 16, 17, and 18 show the experimental results. In these figures, the x-axis represents the number of RT VMs. Figure 16 shows the 99th percentiles of the Critical RT response times, Figure 17 shows the total throughput of the non-RT Heavy Receivers, and Figure 18 shows the achievable CPU utilization.

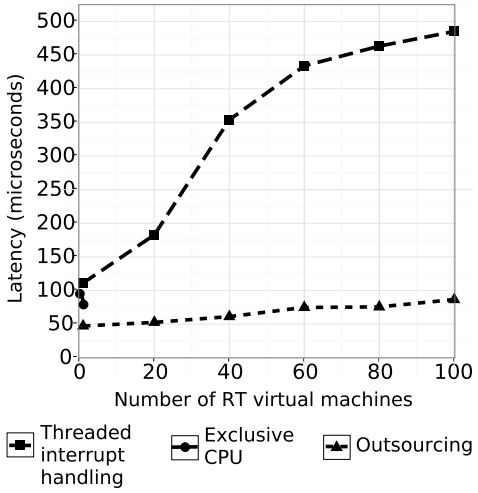

FIGURE 16. Critical RT response time's 99th percentiles when scaling the number of RT virtual machines.

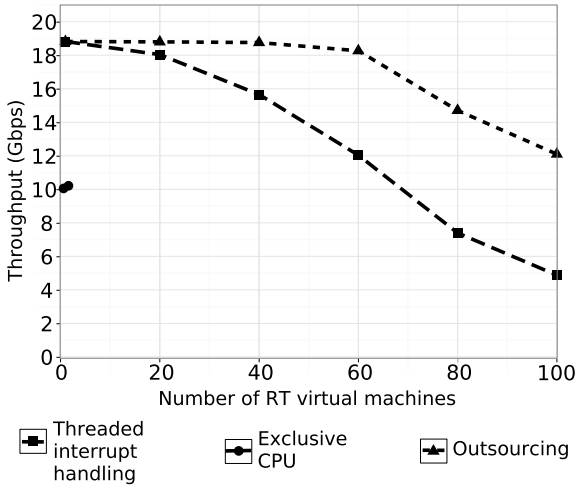

FIGURE 17. Total throughput of Heavy Receiver when scaling the number of RT virtual machines.

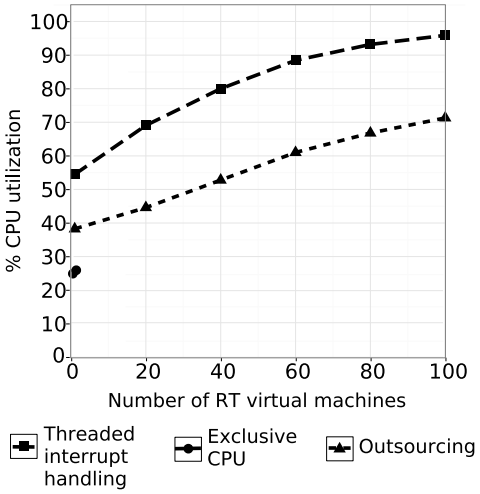

FIGURE 18. Achievable CPU utilization when scaling the number of RT virtual machines.

As shown in Figure 16, the exclusive CPU method did not scale. Both the threaded interrupt handling method and the outsourcing method scaled up to 100 VMs. The outsourcing method produced smaller latency variances of the Critical RT servers than the threaded interrupt handling method. As shown in Figures 17 and 18, the outsourcing method achieved higher throughputs of the Heavy Receivers and lower CPU utilization than the threaded interrupt handling method. Furthermore, the outsourcing method maintained the total throughput of the Heavy Receivers up to 40 VMs, as shown in Figure 17.

V. RELATED WORK

We described two current production RT methods in Sections II-B and II-D. Rostedt and Hart [5], van Riel [8], Christofferson [16], and Crespo *et al.* [17] discussed the threaded interrupt handling method and the exclusive CPU method. We compared their performance with that of the outsourcing method in Section IV. In this section, we discuss other related work.

Improving network stacks of commodity operating systems is an important issue at any time as these systems evolve. Lazy receiver processing (LRP) [29] delays the interrupt handling of receiving processes in the 4.4 BSD OS until these receiving processes are scheduled according to the priority of the processes. The work in [30] prioritizes interrupt handling of Solaris for consistent latency of asynchronous transfer mode (ATM) networks. Some techniques enable user space applications to send and receive packets by polling mode and eliminate overheads of interrupt handling in operating systems. Polling threads [31] and Netmap [32] are examples of such user space I/O techniques. The data plane development kit (DPDK) is used not only for implementing

network appliances, such as routers and content delivery networks (CDNs) but also for fast user space I/O in hosted virtual machine environments [33]. These efforts for improving network stacks have different goals and trade-offs between throughput and CPU utilization as well as latency and variance in latency. Required modifications to the respective commodity operating systems are also different.

To reduce host-guest memory copying in VM environments, some researchers proposed techniques based on memory sharing [34]–[36]. Although these approaches increase the throughput by avoiding message copying and reducing the number of VM context switches, they do not consider latency and latency variance.

The threaded interrupt handling and exclusive CPU methods are the favorite choices for adding RT support to current commodity OSs. RTLinux [37] and Time-Sensitive Linux (TSL) [38] are early examples, and Xenomai [39] is a more recent one. While these systems effectively reduced latency variance, the persistent evolution of the base operating system code introduces new sources of variance. In this paper, we tackle new sources of variance, the priority inversion in the softirq handling, and cache pollution, as described in Sections II-C and II-E.

Socket outsourcing and similar techniques [6], [40]–[42] offload guests' high-level socket operations to the host. These techniques improve throughput by eliminating message copying and by sending TCP acknowledgment packets efficiently. XWAY [43] and XenVMC [44] implement a similar technique for co-located VMs in the Xen hypervisor. In these systems, the guest kernel intercepts socket calls. If the receiver is a co-located VM, messages are sent through shared memory. We adopted socket outsourcing to eliminate message copying and mitigate the cache pollution problem caused by the non-RT servers. Furthermore, we eliminated virtual interrupt handling, as described in Section III-C.

The proposals in [45]–[47] use advanced hardware facilities to improve mainly I/O throughputs in VM environments. This paper proposes a software-based method for consistent latency and compares it with other software-based methods for the same goal. Using software-based methods and advanced hardware facilities together for consistent latency is an interesting research topic for future work.

For realizing real-time characteristics at the network level, proposals such as those of [9], [48], and [49] have been proposed to control the bandwidth according to the priorities of the tenants. To avoid performance degradation and unpredictability, other proposals advocate using bare-metal clouds [50]–[52]. In this study, we eliminated priority inversion problems in complex network stacks in hosted virtual execution environments.

VI. CONCLUSION

In this paper, we described the outsourcing method, which is a new approach for implementing real-time (RT) network stacks in a Linux KVM-based hosted environment. We evaluated the outsourcing method by comparing it with two representative production methods: the threaded interrupt handling method and the exclusive CPU method.

In this study, we ran a high priority RT server and low priority non-RT servers together using these RT methods and analyzed the host and guest kernels. First, we found that vanilla Linux as a host of virtual machines had a priority inversion between RT user processes and non-RT interrupt handling, and this introduced latency variance to the network protocol stack. Next, we confirmed that the threaded interrupt handling method, which makes use of the PREEMPT_RT patch, can eliminate this priority inversion. However, we found that there existed another priority inversion in the softirq handling by the host kernel. Next, we showed that the exclusive CPU method, which allocates an exclusive CPU to a group of RT threads, can remove the priority inversion in softirq handling. However, this method has disadvantages: low utilization of the exclusive processor and low total throughput of co-located non-RT servers. Furthermore, these existing RT methods have the cache pollution problem caused by co-located non-RT servers.

In this paper, we have shown that the outsourcing method can achieve consistent RT latency in a commodity virtual machine environment. This method consists of two techniques along with the existing PREEMPT_RT patch. The first technique divides the softirq handling into RT and non-RT types and removes the priority inversion in softirq handling by the host kernel. The second technique delegates the operations of a guest's network stack to the host of the virtual machine, and thus mitigates the cache pollution problem and removes the priority inversion in softirq handling of a guest kernel.

Compared to the threaded interrupt handling method, the outsourcing method reduced the standard deviation of the latencies of a simple RT server by a factor of 6. At the same time, the outsourcing method improved the throughputs of non-RT servers by up to 5.6% with 32% lower CPU utilization. Compared to the exclusive CPU method, the outsourcing method reduced the standard deviation by a factor of 2 and avoided low utilization of the exclusive CPU. Moreover, the outsourcing method was effective for running two time-sensitive applications: a Voice-over-IP (VoIP) server and a key-value store server.

In this paper, we have shown that the outsourcing method has scalability in terms of the number of RT virtual machines. A four-CPU host was able to execute 40 RT VMs of simple RT servers using the outsourcing method while maintaining the throughputs of non-RT servers. The outsourcing method was more scalable than the threaded interrupt handling method and the exclusive CPU method.

In the future, we would like to use advanced hardware techniques together with the outsourcing method. For example, we would like to use Intel's Cache Allocation Technology (CAT) [18] to further avoid the cache pollution problem. We are also interested in evaluating partitioned RT softirq handling in non-virtualized environments and container-based virtual environments.

ACKNOWLEDGMENT

This paper was presented at the SCF International Conference on Cloud Computing in Seattle, WA, USA, in 2018.

REFERENCES

[1] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013.

[2] J. Park, Q. Wang, J. Li, C.-A. Lai, T. Zhu, and C. Pu, "Performance interference of memory thrashing in virtualized cloud environments: A study of consolidated n-tier applications," in *Proc. 9th IEEE Int. Conf. Cloud Comput. (CLOUD)*, Jun./Jul. 2016, pp. 276–283.

[3] C. Pu *et al.*, "The millibottleneck theory of performance bugs, and its experimental verification," in *Proc. 37th IEEE Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2017, pp. 1919–1926.

[4] Q. Wang, C.-A. Lai, Y. Kanemasa, S. Zhang, and C. Pu, "A study of long-tail latency in n-tier systems: RPC vs. Asynchronous invocations," in *Proc. 37th IEEE Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2017, pp. 207–217.

[5] S. Rostedt and D. V. Hart, "Internals of the RT patch," in *Proc. Linux Symp.*, 2007, pp. 161–172.

[6] H. Eiraku, Y. Shinjo, C. Pu, Y. Koh, and K. Kato, "Fast networking with socket-outsourcing in hosted virtual machine environments," in *Proc. ACM Symp. Appl. Comput. (SAC)*, 2009, pp. 310–317.

[7] O. Garcia, Y. Shinjo, and C. Pu, "Implementation and comparative evaluation of an outsourcing approach to real-time network services in commodity hosted environments," in *Proc. SCF Int. Conf. Cloud Comput. (CLOUD)*, 2018, pp. 189–205.

[8] R. van Riel, "Real-time KVM from the ground up," in *Proc. KVM Forum*, 2015. [Online]. Available: https://wiki.linuxfoundation.org/_media/realtime/events/rt-summit2016/kvm_rik-van-riel.pdf

[9] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: Predictable message latency in the cloud," in *Proc. ACM Conf. Special Interest Group Data Commun. (SIGCOMM)*, 2015, pp. 435–448.

[10] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica, "HUG: Multi-resource fairness for correlated and elastic demands," in *Proc. 13th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2016, pp. 407–424.

[11] P. Xiong, H. Hacigumus, and J. F. Naughton, "A software-defined networking based approach for performance management of analytical queries on distributed data stores," in *Proc. ACM Int. Conf. Manage. Data (SIGMOD)*, 2014, pp. 955–966.

[12] C. Werner, C. Buschmann, T. Jäcker, and S. Fischer, "Bandwidth and latency considerations for efficient SOAP messaging," *Int. J. Web Services Res.*, vol. 3, no. 1, pp. 49–67, 2006.

[13] NAPI. *The Linux Foundation*. Accessed: Jul. 23, 2018. [Online]. Available: https://wiki.linuxfoundation.org/networking/napi

[14] Intel Corporation. *Intel Ethernet Converged Network Adapter X520 Product Brief*. Accessed: Jul. 23, 2018. [Online]. Available: https://www.intel.com/content/www/us/en/ethernet-products/converged-network-adapters/ethernet-x520-server-adapters-brief.html

[15] Steven and D. Rostedt. *KernelShark—A Front end Reader of trace-CMD*. Accessed: Jun. 10, 2018. [Online]. Available: http://rostedt.homelinux.com/kernelshark/

[16] M. Christofferson, "4 ways to improve performance in embedded Linux systems," in *Proc. Korea Linux Forum*, 2013. [Online]. Available: https://events.static.linuxfound.org/sites/events/files/slides/Four%20Ways%20to%20Improve%20Embedded%20Linux%20Performance%20-%20KLF%202013.pdf

[17] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned embedded architecture based on hypervisor: The XtratuM approach," in *Proc. IEEE Eur. Dependable Comput. Conf. (EDCC)*, Apr. 2010, pp. 67–72.

[18] Intel Corporation. *Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family*. Accessed: Jul. 26, 2018. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology

[19] ARM Limited. *ARM 9 Technical Reference Manual–Cache Lockdown*. Accessed: Aug. 21, 2018. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0201d/I33878.html

[20] R. Jones. (1996). *Netperf*. Accessed: Jun. 3, 2018. [Online]. Available: https://hewlettpackard.github.io/netperf

[21] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. (2005). *Iperf: The TCP/UDP bandwidth measurement tool*. Accessed: Jun. 3, 2018. [Online]. Available: http://iperf.sourceforge.net

[22] Endace Technology Limited. *Endace DAG10X2-S Datasheet*. Accessed: Jun. 3, 2018. [Online]. Available: https://www.endace.com/dag-10x2-s-datasheet.pdf

[23] The Linux Foundation. *Perf: Linux Profiling With Performance Counters*. Accessed: Jun. 10, 2018. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[24] Kamailio SIP Server. *The Kamailio SIP Server Project*. Accessed: Jun. 3, 2018. [Online]. Available: https://www.kamailio.org/w

[25] J. Rosenberg *et al.*, *SIP: Session Initiation Protocol*, RFC Standard 3261, Internet Engineering Task Force, Fremont, CA, USA, 2002. [Online]. Available: https://tools.ietf.org/html/rfc3261

[26] R. Gayraud *et al.* (2014). *SIPp*. Accessed: Jun. 3, 2018. [Online]. Available: http://sipp.sourceforge.net

[27] (2015). *Memcached—A Distributed Memory Object Caching System*. Accessed: Jun. 3, 2018. [Online]. Available: https://memcached.org

[28] M. Zhuang and B. Aker. *Load Testing and Benchmarking a Server*. Accessed: Jun. 3, 2018. [Online]. Available: http://libmemcached.org/libMemcached.html

[29] P. Druschel and G. Banga, "Lazy receiver processing (LRP): A network subsystem architecture for server systems," in *Proc. 2nd USENIX Symp. Oper. Syst. Design Implement.*, 1996, pp. 261–275.

[30] F. Kuhns, D. C. Schmidt, and D. L. Levine, "The design and performance of a real-time I/O subsystem," in *Proc. 5th IEEE Real-Time Technol. Appl. Symp.*, Jun. 1999, pp. 154–163.

[31] J. Liu and B. Abali, "Virtualization polling engine (VPE): Using dedicated CPU cores to accelerate I/O virtualization," in *Proc. 23rd ACM Int. Conf. Supercomput.*, 2009, pp. 225–234.

[32] L. Rizzo, "Netmap: A novel framework for fast packet I/O," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2012, pp. 101–112.

[33] The Linux Foundation. *DPDK–Vhost Library*. Accessed: Aug. 21, 2018. [Online]. Available: https://doc.dpdk.org/guides/prog_guide/vhost_lib.html

[34] H. R. Mohebbi, O. Kashefi, and M. Sharifi, "Zivm: A zero-copy inter-VM communication mechanism for cloud computing," *Comput. Inf. Sci.*, vol. 4, no. 6, pp. 18–27, 2011.

[35] C. Pinto, B. Reynal, N. Nikolaev, and D. Raho, "A zero-copy shared memory framework for host-guest data sharing in KVM," in *Proc. IEEE Conf. Ubiquitous Intell. Comput., Adv. Trusted Comput. Scalable Comput. Commun. (ScalCom)*, Jul. 2016, pp. 603–610.

[36] F. Ning, C. Weng, and Y. Luo, "Virtualization I/O optimization based on shared memory," in *Proc. IEEE Int. Conf. Big Data*, Oct. 2013, pp. 70–77.

[37] A. Barabanov and V. Yodaiken, "Introducing real-time Linux," *Linux J.*, vol. 34, p. 9, Feb. 1997.

[38] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole, "Supporting time-sensitive applications on a commodity OS," in *Proc. 5th USENIX Symp. Oper. Syst. Design Implement.*, 2002, pp. 165–180.

[39] P. Gerum. (2004). *Xenomai—Implementing a RTOS Emulation Framework GNU/Linux*. Accessed: Jun. 5, 2018. [Online]. Available: http://www.xenomai.org/documentation/xenomai-2.5/pdf/xenomai.pdf

[40] S. Gamage, R. R. Kompella, D. Xu, and A. Kangarlou, "Protocol responsibility offloading to improve TCP throughput in virtualized environments," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, 2013, Art. no. 7.

[41] A. Nordal, Å. Kvalnes, and D. Johansen, "Paravirtualizing TCP," in *Proc. 6th ACM Int. Workshop Virtualization Technol. Distrib. Comput. Date (VTDC)*, 2012, pp. 3–10.

[42] J. Nakajima *et al.*, "Optimizing virtual machines using hybrid virtualization," in *Proc. ACM Symp. Appl. Comput.*, 2011, pp. 573–578.

[43] K. Kim, C. Kim, S.-I. Jung, H.-S. Shin, and J.-S. Kim, "Inter-domain socket communications supporting high performance and full binary compatibility on Xen," in *Proc. 4th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ. (VEE)*, 2008, pp. 11–20.

[44] Y. Ren *et al.*, "A fast and transparent communication protocol for co-resident virtual machines," in *Proc. 8th IEEE Int. Conf. Collaborative Comput., Netw., Appl. Worksharing (CollaborateCom)*, Oct. 2012, pp. 70–79.

[45] A. Gordon, N. Har'El, A. Landau, M. Ben-Yehuda, and A. Traeger, "Towards exitless and efficient paravirtual I/O," in *Proc. 5th ACM Annu. Int. Syst. Storage Conf. (SYSTOR)*, 2012, Art. no. 10.

[46] A. Gordon *et al.*, "ELI: Bare-metal performance for I/O virtualization," in *Proc. 17th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2012, pp. 411–422.

[47] C.-C. Tu, M. Ferdman, C.-T. Lee, and T.-C. Chiueh, "A comprehensive implementation and evaluation of direct interrupt delivery," in *Proc. 11th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ. (VEE)*, 2015, pp. 1–15.
[48] M. P. Grosvenor *et al.*, "Queues don't matter when you can jump them!" in *Proc. 12th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2015, pp. 1–14.
[49] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "FairCloud: Sharing the network in cloud computing," in *Proc. ACM SIGCOMM Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2012, pp. 187–198.
[50] P. Rad, A. T. Chronopoulos, P. Lama, P. Madduri, and C. Loader, "Benchmarking bare metal cloud servers for HPC applications," in *Proc. IEEE Int. Conf. Cloud Comput. Emerg. Markets (CCEM)*, Nov. 2015, pp. 153–159.
[51] Y. Omote, T. Shinagawa, and K. Kato, "Improving agility and elasticity in bare-metal clouds," in *Proc. 12th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2015, pp. 145–159.
[52] S. Crago *et al.*, "Heterogeneous cloud computing," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2011, pp. 378–385.

OSCAR F. GARCIA received the B.S. degree in systems and computation engineering from the Universidad de Los Andes, Colombia, in 2008, and the M.S. degree from the University of Tsukuba, Japan, in 2015, where he is currently pursuing the Ph.D. degree with the Graduate School of Systems and Information Engineering. His research interests include operating systems, real-time systems, cloud computing, and visual programming languages.

YASUSHI SHINJO (S'91–M'93) received the Ph.D. degree from the University of Tsukuba in 1993. He is currently an Associate Professor with the Department of Computer Science, University of Tsukuba. He is interested in the areas of operating systems, parallel and distributed computing, security, privacy, decentralized social networking services, and virtual systems. He has authored papers in these areas. His research has been supported by the Ministry of Education, Culture, Sports, Science and Technology (MEXT), Japan, and the Japan Science and Technology Agency. He is a member of the ACM, IPSJ, and Japan Society for Software Science and Technology. He has been serving on the Program Committee for the IEEE International Conference on Cloud Computing since 2015.

CALTON PU (S'81–M'87–SM'05–F'16) received the Ph.D. degree from the University of Washington. He was a Faculty Member with Columbia University and the Oregon Graduate Institute. He is currently a Professor and the John P. Imlay, Jr. Chair of software with the College of Computing, Georgia Institute of Technology. He is involved in several projects in systems and database research. He has authored over 70 journal papers and book chapters, and 270 conference and refereed workshop papers. His recent research has focused on big data in Internet of Things, automated N-tier application performance, and denial of information. He is a fellow of AAAS. He served for more than 120 program committees. He served as a General (Co-)Chair and a PC (Co-)Chair for more than 20 times.