Tulips R
University of Tsukuba Repository

Research on real-time network stacks of commodity hosted virtual machine environments

著者（英）	Fernando Garcia Alvarado Oscar
year	2019
その他のタイトル	実用ホスト型仮想計算機環境の実時間ネットワーク・スタックに関する研究
学位授与大学	筑波大学 (University of Tsukuba)
学位授与年度	2018
報告番号	12102甲第8995号
URL	http://doi.org/10.15068/00156283

Research on real-time network stacks of commodity hosted virtual machine environments

March 2019

Oscar Fernando Garcia Alvarado

Research on real-time network stacks of commodity hosted virtual machine environments

Graduate School of Systems and Information Engineering

University of Tsukuba

March 2019

Oscar Fernando Garcia Alvarado

Abstract

It is a common practice to run real-time and time-sensitive applications on commodity OSs, such as Linux. Commodity OSs cannot provide sufficient real-time capabilities for some real-time applications. To address this problem, developers and researchers are developing small extensions or patches to commodity OSs. This thesis achieves a consistent real-time (RT) response time in commodity virtual machine (VM) environments which have have longer and more complex network protocol stacks.

The target hosted virtual machine environment hosts RT and non-RT network servers in VMs. The research objective of this thesis is to achieve the following goals at the same time: 1) Achieve short and consistent latency for RT servers. 2) Obtain high throughput for non-RT servers and avoid low CPU utilization within the bound of the consistent latency for RT servers.

To achieve these goals, this thesis analyzed the message processing path of RT and non-RT servers in vanilla Linux and two conventional RT methods. First, the author has confirmed a priority inversion in the interrupt-first host kernel of vanilla Linux. The author has fpund two new sources of variances: a priority inversion in softirq handling and the cache pollution by co-located non-RT servers.

To solve these problems, the author proposes a new approach to an RT network stack in a Linux KVM-based virtual machine environment. This approach is called the "socket outsourcing with partitioned RT softirq handling" method or the outsourcing method for short. The author addresses the priority inversion problem in the host's softirq handling by dividing softirq handling into RT and non-RT types. The author mitigates the cache pollution problem and prevents the priority inversion problem in a guest's softirq handling by extending socket outsourcing. Socket outsourcing allows a guest kernel to delegate high-level network operations to the host kernel. When a guest process invokes a socket operation, its processing is delegated to the host. This removes the duplicated message copying in conventional hosted virtualization and reduces the cache pollution by non-RT servers.

This thesis evaluates the proposed method by comparing to two conventional RT methods. 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. Moreover, the outsourcing method was effective for running two time-sensitive applications: a Voice-over-IP (VoIP) server and a key-value store server. The outsourcing method was more scalable in terms of the number of RT VMs. 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.

Contents

1 Introduction **1**

2 Related Work **7**

2.1 Adding real-time capabilities to Commodity OSs 7

2.1.1 Early proposals for 4.4 BSD and Solaris 7

2.1.2 Early proposals of adding real-time capabilities to Linux . . . 10

2.1.3 The PREEMPT_RT patch for Linux 14

2.2 RT virtual machines 14

2.2.1 KVM for NFV 15

2.2.2 RT-Xen 15

2.2.3 Linux Jailhouse 16

2.2.4 Leulo 17

2.3 Improving network throughput of virtual machines 17

2.3.1 Socket outsourcing 17

2.3.2 Accelerating host-guest message passing 18

2.4 Resource reservation and service level objectives 19

2.4.1 Exclusive physical resources allocation 19

2.4.2 Heracles 19

2.4.3 Silo 20

2.4.4 IRMOS/Real-time SOIs 21

2.5 Network I/O without interrupt handling 21

2.6 Network I/O using advanced hardware features 23

2.6.1 Virtual machine device queues (VMDQs) and Single-root input/output virtualization (SR-IOV) 23
2.6.2 Cache Allocation Technology (CAT) and vCAT 25
2.7 Real-time networks 25
2.8 Network stacks for high-performance computing 26

3 Analyzing vanilla Linux and two conventional RT methods 28
3.1 Experimental environment 29
3.2 Vanilla Linux and two conventional RT methods 31
3.3 Latency and throughput in vanilla Linux and the conventional RT methods 35
3.4 Measuring latencies of network stack components using Ftrace and light-weight probes 38
3.4.1 Processing path analysis with lightweight probes 39
3.4.2 Analyzing the message processing path of the Critical RT server with lightweight probes 39
3.4.3 Finding priority inversions at the "host receive" segment in the threaded interrupt handling method with Ftrace and Kernelshark 43
3.5 Cache pollution by co-located non-RT Servers 45
3.6 Summary of analyzing the network stack of a hosted virtual machine environment 47

4 Partitioned RT softirq handling 49
4.1 Interrupt handling in partitioned RT softirq handling 50
4.2 Implementation of partitioned RT softirq handling 52
4.2.1 Modifying the NAPI module 52
4.2.2 Modifying the softirq mechanism 55

5 RT socket outsourcing 57
5.1 Conventional socket outsourcing 57
5.2 RT socket outsourcing 60

5.3 Implementation details of RT socket outsourcing 63
5.3.1 The guest client module 63
5.3.2 The Host server module 65
5.3.3 The extended idle process 66

6 Experimental evaluation 69

6.1 Experimental setup for running a simple RT server 69
6.2 Experimental results using a simple RT server 72
6.3 Effects of individual techniques 76
6.4 Processing path analysis with lightweight probes 78
6.5 Cache pollution in the RT methods 82
6.6 Message processing paths of non-RT server 83
6.7 Application benchmarks 84
6.7.1 A voice-over-IP (VoIP) server. 84
6.7.2 Memcached 87
6.8 Scalability of RT virtual machines 89
6.9 Using partitioned RT softirq handling in container-based virtualization 93
6.10 Current restrictions and limitations 94

7 Conclusion 96

Acknowledgments 102

List of publications 103

List of Figures

1-1 Development race of mainline and RT patch. 3

1-2 The target virtual machine environment that executes RT and non-RT servers together. 4

2-1 Comparing the network subsystems of 4.4 BSD and LRP. 8

2-2 Prioritized interrupt handling in Solaris. 9

2-3 RT-Linux running the Linux kernel as an RT task of an RTOS in RT-Linux. 10

2-4 The architectures of Xenomai and RTAI. 11

2-5 Network subsystem in TimeSys Linux. 13

2-6 Architecture of Linux Jailhouse. 16

2-7 Comparison of a typical network stack and that of Virtualization Polling Engine. 22

2-8 Comparison of a typical network stacks and a network stack using SR-IOV. 24

3-1 Running an RT server and co-located non-RT servers in a target virtual machine environment. 29

3-2 The components of the network stack in the target environment. . . . 30

3-3 Interrupt handling in vanilla Linux. 32

3-4 Interrupt handling using the threaded interrupt handling method. . . 33

3-5 Interrupt handling using the exclusive CPU method. 35

3-6 Distribution of the Critical RT server's response times in vanilla Linux and the two conventional RT methods without Heavy Receivers. . . . 36

3-7 Distribution of the Critical RT server's response times in vanilla Linux and the two conventional RT methods with Heavy Receivers. 36
3-8 Total throughput of Heavy Receivers. 37
3-9 Achievable CPU utilization. 38
3-10 Division of the message processing path into three segments. 39
3-11 Latencies of three segments of the message processing path without running Heavy Receivers. 41
3-12 Latencies of three segments of the message processing path with running Heavy Receivers. 42
3-13 Priority inversion in the softirq handling in the host OS using the threaded interrupt handling method. 44
3-14 99th percentile latencies of the Critical server in the "guest" segment at different request's inter-arrival times. 46

4-1 Interrupt handling using the outsourcing method. 50
4-2 The trace of interrupt handling in in partitioned RT softirq handling. 51
4-3 Adding rt_poll_list to the softnet_data structure. 53
4-4 Adding sysctl parameter net.core.rtnet_prio. 53
4-5 Modifying the function napi_schedule_irqoff(). 54
4-6 Adding a new softirq kind "RT_NET_RX_SOFTIRQ" 55

5-1 Comparison of network paravirtualization in the threaded interrupt handling method and conventional socket outsourcing. 59
5-2 Comparison between conventional socket outsourcing and RT socket outsourcing. 61
5-3 Implementation of the `recvfrom()` system call in a guest. 64
5-4 Implementation of the `skhst_recvfrom()` function in the host. . . . 66
5-5 Implementation of the `notify_guest()` function in the host. 67
5-6 The extended idle process 68

6-1 The experimental environment. 70

6-2 Distribution of the Critical RT server response times without running Heavy Receivers. 74
6-3 Distribution of the Critical RT server response times with Heavy Receivers. 75
6-4 Total throughput of Heavy Receivers. 76
6-5 Achievable CPU utilization. 76
6-6 Distribution of the Critical RT server response times with the outsourcing method using individual techniques. 77
6-7 Division of the message processing path into three segments. 78
6-8 Latencies in three segments of the processing path of RT messages without running Heavy Receivers. 80
6-9 Latencies in three segments of the processing path of RT messages with running Heavy Receivers. 81
6-10 LLC miss ratio of the RT threads. 82
6-11 Message processing path of a non-RT Heavy Receiver. 83
6-12 The results of application benchmark using a VoIP server. 86
6-13 The results of application benchmark using memcached. 88
6-14 Scaling the number of RT virtual machines (Netperf as critical RT server). 90
6-15 Scaling the number of RT virtual machines (VoIP server as critical RT server). 91
6-16 Scaling the number of RT virtual machines (Memcached as critical RT server). 92
6-17 Distribution of the Critical RT server in container-based virtualization. 93

List of Tables

3.1 Statistical values of the Critical RT server response times in vanilla Linux and the two conventional RT methods without Heavy Receivers (microseconds). 36

3.2 Statistical values of the Critical RT server response times in vanilla Linux and the two conventional RT methods with Heavy Receivers (microseconds). 36

3.3 Summary of the RT methods. 48

6.1 Specifications of the machines and their active cores in the experiments. 71

6.2 Scheduling policy and priority of the threads in the host OS. 72

6.3 Statistical values of the Critical RT server response times without running Heavy Receivers (microseconds). 74

6.4 Statistical values of the Critical RT server response times with Heavy Receivers (microseconds). 75

6.5 Statistical values of the Critical RT server with the outsourcing method using individual techniques. 77

6.6 Statistical values of the Critical RT server response times with running Heavy Receivers in container-based virtualization (microseconds). . . 93

Chapter 1

Introduction

Real-time and time-sensitive systems are everywhere from surroundings and offices to data centers. Simple real-time systems are embedded in home electrical appliances, such as microwave ovens and kids toys. People enjoy video streaming everyday with video players in PCs. Embedded systems are becoming complex. Examples of complex embedded systems are automotive navigation systems, medical imaging systems, and robotic assembly lines in factories. When they are connected to the Internet, they construct Internet of Things (IoT). Data centers host time-sensitive network servers, such as voice-over IP (VoIP) servers and web search engines.

Many real-time applications are built on Real-Time Operating Systems (RTOSs). Representative RTOSs are FreeRTOS [8], ITRON [68], Micrium μC/OS [87], and VxWorks [93]. Recently, it is a common practice to run real-time and time-sensitive applications on commodity OSs, such as Linux. Especially complex real-time systems are often built on commodity OSs because commodity OSs provide rich networking and graphics APIs.

In this thesis, we define *real-time* applications that run on commodity OSs as follows.

- They use the RT extensions, such as POSIX.1b (IEEE 1003.1b-1993) [39]. These extensions include priority scheduling, real-time signals, semaphores, and memory locking.

- They work well in non-virtualized or *native* commodity OSs.

Commodity OSs can provide sufficient real-time capabilities for some real-time applications with fast CPUs. However, commodity OSs cannot provide sufficient real-time capabilities for some other real-time applications even with fast CPUs. To address this problem, developers and researchers are developing small extensions or patches to commodity OSs. Early examples of such patches are Lazy receiver processing (LRP) for 4.4 BSD [24], prioritized interrupt handling for Solaris [53], and RT-Linux [5], Time-Sensitive Linux [34], and Resource kernel [33] for Linux. Currently, one of the most active patches is PREEMPT_RT patch [84] for Linux.

Commodity OSs continuously evolve and become complex. For example, the size of the Linux kernel was 8,000,000 in source lines of code (SLOC) in 2008. It is 20,000,000 in 2018. These changes are not only for adding new features but also for improving performance. For example, in 2013, Linux 3.11 incorporated a polling mechanism in a new network API called NAPI [25] to the network stack. In 2017, Linux 4.9 incorporated a TCP algorithm called Bottleneck Bandwidth and Round-trip propagation time (BBR) [12].

There is an inherent problem in the continuous evolution of commodity OSs. That is, most of these changes favor throughput over latency and variance of latency . For example, NAPI improves the throughput by reducing the interrupt overhead and performs a fair network processing among devices at the cost of high variances [25].

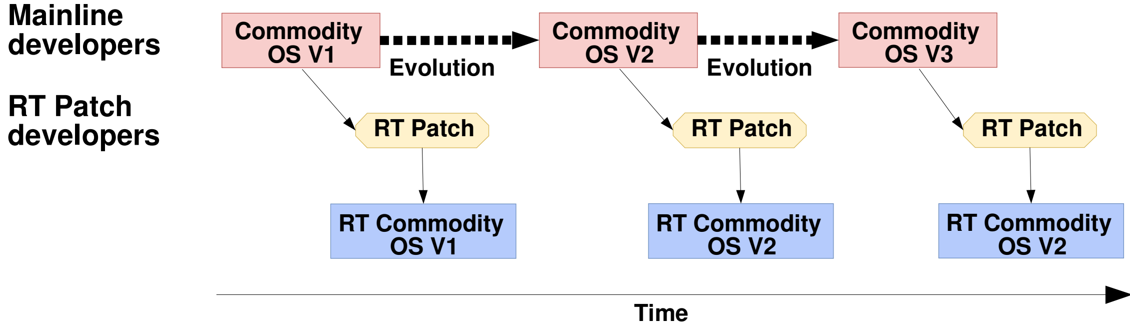


Figure 1-1: Development race of mainline and RT patch.

Because improving real-time capabilities of a commodity OS often decreases throughput, it is usually developed by a separated group as a patch. Figure 1-1 shows that the mainline group of developers releases new versions of the commodity OS for improving throughput. At the same time, the group of RT developers makes an RT patch for each version. This development race is persistent from earlier systems, such as LRP, to current systems, such as PREEMPT_RT patch. In this thesis, we choose a crucial problem from this development race. That is, we achieve a real-time network stack in virtual machine environments.

Figure 1-2 presents a target hosted virtual machine environment of this thesis. This environment hosts RT and non-RT network servers in VMs. This type of an environment is also known as a mixed criticality system [9,11]. RT servers are critical servers and require short and consistent response times. Consistent response times mean low variance or jitter of response times [17]. Non-RT servers wish high throughputs. The CPUs and memory of the host machine are shared by both RT and non-RT servers. The RT servers use an RT network, whereas the non-RT servers use a non-RT network. The network interface cards (NICs) connected to these networks are referred to as an RT NIC and a non-RT NIC, respectively. We assume that the delay and bandwidth of the RT network are guaranteed by using the methods described by [15, 45, 92, 96]. The non-RT network is a best-effort network.

When we run RT servers and non-RT servers together in the target 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. It is

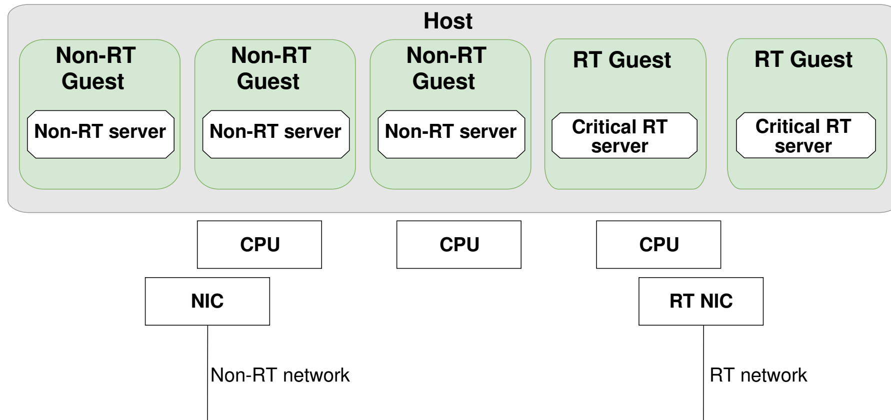


Figure 1-2: The target virtual machine environment that executes RT and non-RT servers together.

not trivial to find the causes of response time problems because the network stack of the target environment is complex and evolving.

It is known that using the PREEMPT_RT patch is not sufficient for realizing real-time network stacks in virtual machine environments. However, its reason was not clear. Most existing systems do not solve this problem but bypass the problem. They allocate exclusive physical resources to the threads of the RT servers [16,18,91]. However, this sacrifices CPU utilization. Several studies show the low CPU utilization ranging between 7% and 50% in most data centers [13,22,55,61].

The research objective of this thesis is to achieve the following goals at the same time:

- Achieve short and consistent latency for RT servers.
- Obtain high throughput for non-RT servers and avoid low CPU utilization within the bound of the consistent latency for RT servers.

To achieve these goals, we began with analyzing the message processing path of RT and non-RT servers in vanilla Linux and two conventional RT methods. As a hypervisor, we used the KVM hypervisor which is integrated into the Linux kernel. First, we have confirmed a priority inversion in the interrupt-first host kernel of vanilla Linux. We found two new sources of variances, a priority inversion in softirq mechanism and the cache pollution by co-located non-RT servers. Note that in this

thesis, we use the term "priority inversion" in a general sense as in [49, 53, 69, 101]. If a non-RT task with a lower priority delays the execution of an RT task with a high priority, we call this a priority inversion.

Vanilla Linux has a priority inversion problem in interrupt handling and executes any interrupt handler first, prior to any high-priority processes including threads of the RT servers. One conventional RT method for Linux uses the PREEMPT_RT patch [84], which executes interrupt handlers by threads with their own priorities and addresses this priority inversion problem. We call this method *the threaded interrupt handling method*. Whereas threaded interrupt handling eliminates the first priority inversion problem, we have found that the second one in softirq handling of the host kernel remains. The second conventional RT method allocates an exclusive CPU to a group of host RT threads. We call this method *the exclusive CPU method*. Although the exclusive CPU method bypasses 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.

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. This is an extension of the threaded interrupt handling method and uses the PREEMPT_RT patch to address the priority inversion problem in the interrupt-first host kernel. Next, we address the priority inversion problem in the host's softirq handling by dividing softirq handling into RT and non-RT types. Finally, we mitigate the cache pollution problem and prevent the priority inversion problem in a guest's softirq handling by extending socket outsourcing [27]. Socket outsourcing allows a guest kernel to delegate high-level network operations to the host kernel. When a guest process invokes a socket operation, its processing is delegated to the host. This removes the duplicated message copying in conventional hosted virtualization and reduces the cache pollution by non-RT services. In addition, we remove interrupt handling from a guest kernel and eliminate the priority inversion problem in softirq

handling of the guest kernel.

We evaluate our proposed method by comparing two conventional RT methods. 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. Moreover, the outsourcing method produced better results for running two real-time applications: a Voice-over-IP (VoIP) server and a key-value store server.

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.

Chapter 2

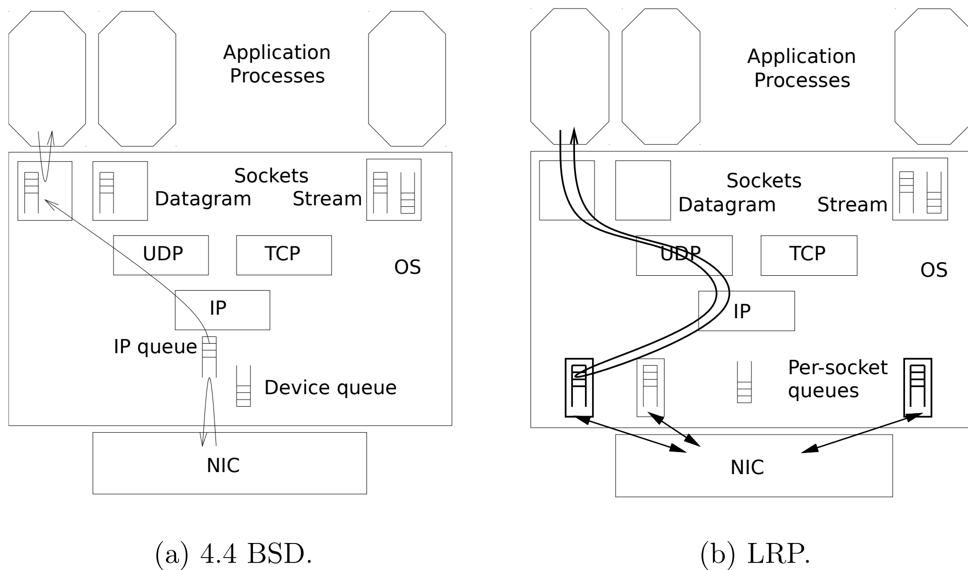
Related Work

This chapter provides related work. The following subsections are structured according to the areas of interest in this thesis. Section 2.1 discusses adding real-time capabilities to commodity OSs. Section 2.2 describes related work in RT virtual machines. Section 2.3 describes previous work in improving network throughput of virtual machines. Section 2.4 presents techniques for guaranteeing Service Level Objectives (SLOs) in real-time applications. Section 2.5 discusses network I/O techniques without interrupt handling. Section 2.6 describes hardware-based techniques. Section 2.7 presents techniques for adding realtimeness in the network level. Finally, Section 2.8 describes related work in network stacks for High Performance Computing (HPC) applications.

2.1 Adding real-time capabilities to Commodity OSs

2.1.1 Early proposals for 4.4 BSD and Solaris

The vanilla kernel of 4.4BSD executes network interrupt handlers first, prior to user processes. Lazy Receiver Processing (LRP) [24] is a network subsystem that schedules the network processing to improve fairness and solve the problem in interrupt handling. LRP delays the interrupt handling of receiving processes until these receiving processes are scheduled according to the priority of the processes.



(a) 4.4 BSD. (b) LRP.

Figure 2-1: Comparing the network subsystems of 4.4 BSD and LRP.

Figure 2-1 compares the traditional network subsystem in 4.4 BSD and the subsystem proposed by LRP. In Figure 2-1a, the interrupt handler of traditional network subsystem of 4.4 BSD puts incoming messages to a single queue, and performs IP and TCP processing for any processes. In contrast, in Figure 2-1b, the network subsystem of LRP has per-socket queues and the NIC can use any per-socket queues. This network subsystem processes incoming messages as follows:

1. The NIC demultiplexes incoming messages based on the destination sockets and puts them into their respective socket queues.

2. When a destination process is scheduled, the kernel performs the IP and TCP processing for this process.

In LRP, the message processing of a low priority process does not preempt the execution of high priority processes. In addition, the network processing does not occur until a user process receives it explicitly through systems calls.

The experiments presented in the LRP paper [24] were performed on a private Asynchronous Transfer Mode (ATM) network between SPARC stations. In terms of throughput, the LRP performance was comparable with the unmodified 4.4 BSD. An experiment presented in the paper measured the latency of a network server when

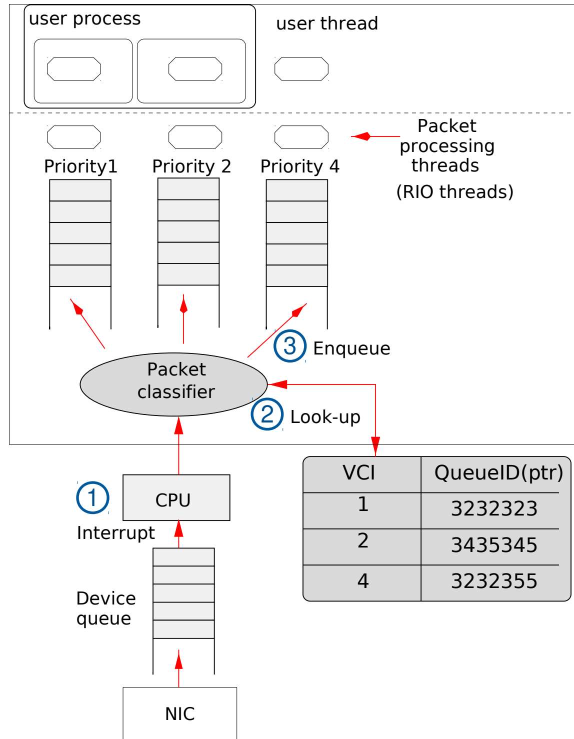


Figure 2-2: Prioritized interrupt handling in Solaris.

the machine of the network server had a high network load. A client and the server performed a round trip latency test using short UDP messages at a fixed rate. In LRP and 4.4 BSD, the measured latency varied with the background traffic rate. However, in 4.4 BSD the latency increased more. This execution time was approximately 60 μs. In LRP, this execution time was approximately 25 μs.

Paper [53] proposes the Real-time I/O (RIO) subsystem for the Solaris 2.5 kernel. Similar as LRP, RIO prioritizes interrupt handling of Solaris for consistent latencies of ATM networks. RIO performs all protocol processings in kernel threads, called RIO threads. RIO threads are scheduled with real-time priorities. Figure 2-2 describes the structure of the RIO subsystem. RIO has a packet classifier that takes advantage of the demultiplexing feature in ATM. The classifier uses the ATM's Virtual Channel Identifier (VCI) field in a request message to determine the RIO thread of its final destination. The RIO threads are co-scheduled with real-time user threads and process I/O requests.

These proposed systems, LRP and RIO, effectively reduced latency and latency

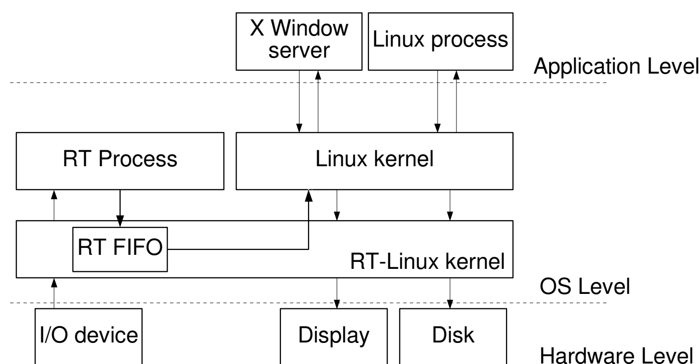


Figure 2-3: RT-Linux running the Linux kernel as an RT task of an RTOS in RT-Linux.

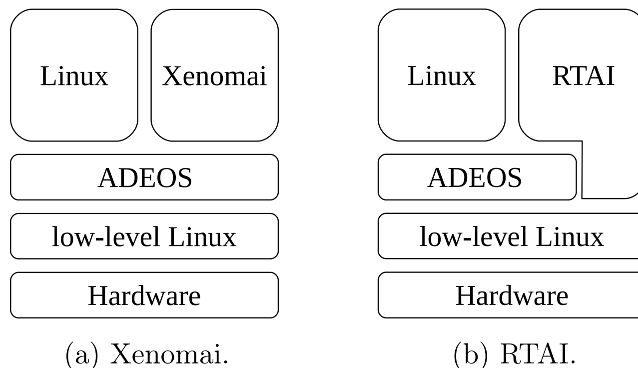
variance in the network stacks of 4.4 BSD and Solaris 2.5, respectively. However, the results are affected by the continuous evolution of the underlying OS code. Furthermore, these proposed systems cannot deal with virtual execution environments. In this thesis, we add real-time capabilities to the network stack of a hosted virtual machine environment. To achieve this, we eliminate new sources of variance, the priority inversion in softirq handling of the host kernel and cache pollution.

2.1.2 Early proposals of adding real-time capabilities to Linux

RT-Linux [5] proposes an RT OS underneath Linux. In RT-Linux, the Linux kernel is a normal task of an RT OS kernel, it runs when there are not runnable RT processes and it is preempted whenever an RT process becomes runnable.

In RT-Linux, every RT process is separated into two processes as shown in Figure 2-3. The first process has hard RT constraints and is defined in a kernel module (identified as RT Process in the figure). The second process executes as a Linux process. The communication between these two processes is done through special queues called real-time FIFOs. The RT process never blocks when it reads or writes a real-time FIFO. In the figure, the RT process copies data from the device into the real-time FIFO. The Linux process reads the data from the other end of the real-time FIFO and displays and stores the data in a file.

This design was taken because a commodity OS does not provide timing guar-



(a) Xenomai. (b) RTAI.

Figure 2-4: The architectures of Xenomai and RTAI.

antees for resuming suspended processes. At that time, this was mostly caused by non-preemptible portions of the kernel of Linux. If an RT process runs in normal Linux and reads a device buffer, the RT process may lose the data. This design of RT-Linux bypasses the Linux kernel and allows high responsiveness without substantial changes to the Linux kernel.

Xenomai [32] and the RealTime Application Interface (RTAI) for Linux [10] use the Adaptive Domain Environment Operating Systems (ADEOS) microkernel to schedule real-time processes while the Linux kernel handles the remaining functionalities. Figure 2-4a describes the architecture of Xenomai. In Xenomai, ADEOS handles hardware interrupts and propagates them in sequence to the Xenomai component. Depending on the destination process, the Xenomai component decides to handle an interrupt or delegate it to the Linux kernel. RTAI uses a different architecture than Xenomai as shown in Figure 2-4b. RTAI intercepts all the interrupts, and the ADEOS microkernel propagates those interrupts of non-RT processes to the Linux kernel. RTAI adopts this approach to avoid the overhead of the ADEOS kernel in the handling of interrupts to real-time processes.

Paper [6] provides a comparative evaluation between Xenomai and RTAI. In a real-time network communication experiment, Xenomai and RTAI outperformed Linux in terms of latency and latency variance. Comparing both approaches, RTAI presented slightly lower latencies and latency variances.

Paper [34] shows that a commodity OS with real-time capabilities requires 1)

fine-grained timers, 2) a preemptible kernel, and 3) RT schedulers. The authors introduce Time-Sensitive Linux (TSL), a Linux-based kernel that realizes these three requirements. For achieving fine-grained timers, TSL implements *firm timers* which implements one-shot timers and *soft timers* [4]. Soft-timers check and fire at special points in the kernel such as systems calls and interrupts. This reduces the number of one-shot timers reprogramming and the overhead of interrupt handling. To reduce the number of non-preemptible areas in the kernel, TLS adopts the result of the Linux preemptible kernel project [64] that allows preemption at anytime the kernel is not holding a spinlock or running an interrupt handler. Finally, TSL implements two RT schedulers to support different types of real-time applications. The first one is a proportion-period CPU scheduler that requires the specification of proportion and period per task. The second one is a priority CPU scheduler which executes the runnable process with the highest priority at any given time.

The PREEMPT_RT patch for Linux incorporates some ideas proposed in TSL Linux. We describe the PREEMPT_RT patch in Section 2.1.3.

Resource kernel [77] separates resource specification from resource management. The kernel can choose the most appropriate resource management scheme to satisfy the demands of applications. This paper proposes a resource reservation model for CPU resources. This model employs a fixed priority scheme where each reservation defines a period T or a deadline D. T is assigned for the rate-monotonic scheduling scheme and D is assigned for the deadline-monotonic scheme.

TimeSys Linux [33] extends Resource kernel, and proposes a resource reservation model for network bandwidth. Figure 2-5 presents the network subsystem of this approach. First, the paper defines a *NetR reserve* that allows applications to specify their network requirements. With a NetR reserve, an RT application can specify the volume of data, the data reception period and the deadline. Each NetR reserve posses its own dedicated backlog queue. The network interrupt handler demultiplexes inbound packets and places them on the respective backlog queues. For other non-RT applications without a NetRT reserve, the interrupt handler places their packets into a default backlog queue. Each NetR reserve spawns a new dedicated NetR thread.

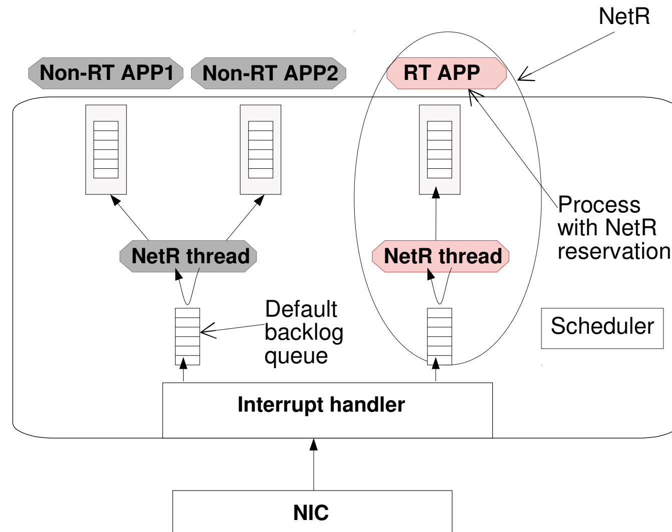


Figure 2-5: Network subsystem in TimeSys Linux.

This thread processes the backlog queue of the NetR reserve. When an application defines a NetR reserve, the application may share the CPU reserve with its NetR thread. A default NetRT thread processes the default backlog queue.

This network subsystem is implemented in TimeSys Linux, a fully preemptive version of Linux that provides a portable resource kernel framework. The experimental results demonstrate the need for control and accounting in the handling of network interrupts. This study also found that incoming traffic from a port may hinder outgoing traffic to another port. The paper proposes allocating a thread for each port to take advantage of CPU reserves.

These systems, RT-Linux, Xenomai, RATI, TS Linux, and resource kernel effectively reduced latency and latency variance in the network stacks of Linux. However, the results are affected by the continuous evolution of the underlying OS code. Furthermore, these proposed systems cannot deal with virtual execution environments. In this thesis, we add real-time capabilities to the network stack of a Linux KVM-based virtual machine environment. To achieve this, we eliminate new sources of variance, the priority inversion in softirq handling of the host kernel and cache pollution.

2.1.3 The PREEMPT_RT patch for Linux

The PREEMPT_RT patch [84] is a project that is officially supported by the Linux Foundation and modifies the Linux kernel to add real-time capabilities. A cause of latency variance in Linux is priority inversion where a high priority thread must wait for releasing a resource that is occupied by a lower priority task. The PREEMPT_RT patch implements threaded interrupt handlers to resolve this issue.

Another cause of latency in Linux is the non-preemptible areas of the kernel. The PREEMPT_RT patch is a successor of the Linux preemptible kernel project [64] and replaces the non-preemptible spinlocks by preemptible sleeping spinlocks. The PREEMPT_RT patch introduces sleeping spinlocks which implement a priority inheritance protocol [84].

The PREEMPT_RT patch for Linux effectively reduced latency and latency variance in the network stacks of Linux. This is a current successful project that collaborates with the mainline developer group of Linux. However, using this patch is not sufficient as discussed in Section 1. Many conventional RT systems bypass the problems in the PREEMPT_RT patch by allocating exclusive CPU resources to a group of RT threads. This sacrifices CPU utilization.

In this thesis, we have found a priority inversion problem in the PREEMPT_RT patch. We address the priority inversion problem by dividing softirq handling into RT and non-RT ones in the host kernel of a virtual machine environment. Furthermore, we eliminate the priority inversion problem in a guest kernel as well by extending socket outsourcing.

2.2 RT virtual machines

In data centers, using Virtual Machines (VMs) is a must. Because each service has a level of criticality, the service can choose an operating system that assures the execution against a failure (e.g. deadlines misses) [11] by using a virtual machine. Using virtual machine eases migration of services among the data-center servers for achieving lower latencies and fault tolerance. In addition, developers and researchers

are working on implementing real-time hypervisors.

2.2.1 KVM for NFV

KVM4NFV [43] is a project that modifies the KVM hypervisor to reduce the interrupt latency variance for data plane Virtual Network Functions (VNFs). KVM4NFV modifies the KVM version of the PREEMPT_RT patch and eliminates spinlocks in the code paths of virtual interrupt delivery and uses non-threaded interrupts to reduce the number of context switches [74].

KVM4NFV focuses only on reducing latency variances by using advanced hardware support such as Single-root Input/Output virtualization (SR-IOV). In this thesis, we propose a software-based method for consistent latency and compares it with other software-based methods for the same goal.

2.2.2 RT-Xen

The base scheduler of the Xen hypervisor is called Credit Scheduler [14], which is highly tuned for achieving fairness among VMs. This scheduler uses 30ms time slices for CPU allocation and is not suitable for running real-time services.

RT-Xen [94] proposes a VM scheduling framework for real-time services in Xen. RT-Xen implements global and partitioned schedulers and both schedulers can support dynamic and static priorities to run VMs. Through experimental evaluation, the authors show that using a partitioned scheduler in a guest OS and a global scheduler in the hypervisor resulted in the best performance when using a periodic server. The global scheduler running VMs as deferrable servers achieved lower deadline miss ratios under overload when compared with other scheduling policies.

In this thesis, we use Linux KVM that uses the scheduler of Linux for scheduling RT and non-RT VMs. Linux has an RT scheduler that realizes POSIX RT extensions [39] as well as a normal scheduler for fairness, called the Completely Fair Scheduler (CFS). The RT scheduler of Linux provides better RT capabilities than that of RT-Xen. Furthermore, in this thesis, we eliminate the priority inversion problems in both

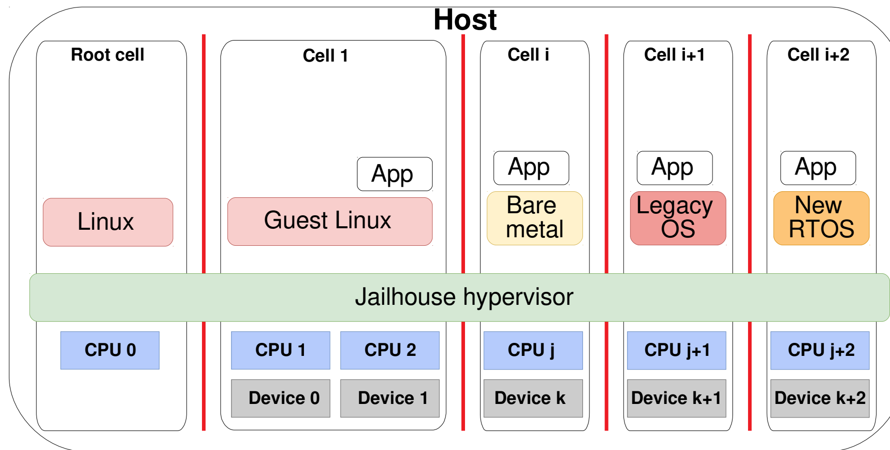


Figure 2-6: Architecture of Linux Jailhouse.

the host kernel and a guest kernel of Linux KVM.

2.2.3 Linux Jailhouse

Jailhouse [88] is a Linux-based partitioning hypervisor which runs bare-bone applications and provides isolation between them. Since Jailhouse lacks advanced resource management functionalities, it does not replace modern hypervisors such as Xen or KVM. Instead, Jailhouse focuses on safe-critical applications, such as Magnetic Resonance Imaging (MRI) devices, train controls and some robot factories utilize high-speed of control systems that must react to events at high rates and require low latencies. Jailhouse is also useful for HPC applications to minimize the interference of other processes in a single machine [51].

Figure 2-6 shows the architecture of Jailhouse. Jailhouse implements an abstraction layer that splits the host into isolated partitions, called cells. Each cell has a set of resources (CPU, memory regions, and PCI devices) and only one guest OS or bare-bone application can run in a cell. The functionality of the Jailhouse hypervisor is limited to maintain the isolation of the cells. The cell with Linux that bootstraps Jailhouse is called the *root cell*. The root cell manages the other cells. The configuration of each cell is static and must be defined before it launches. This configuration defines the hardware that each cell can access.

The goal of Jailhouse differs from the goal of this thesis. The goal of Jailhouse is to achieve only real-time capabilities. On the other hand, our goal is to achieve consistent latency for RT servers and at the same time high throughput for non-RT servers within the bound of the consistent latency for RT servers. Jailhouse does not provide real-time network stacks. In this thesis, we show the implementation of a real-time network stack relying on a real-time network.

2.2.4 Leulo

Leulo [72] is a hypervisor designed to alleviate the overhead in virtualized real-time applications that utilize TCP. The TCP processing in a guest is often delayed and this delay can affect the network performance. For example, delays in handling transmit window updates from a peer can inhibit the transmission of buffered packets [72]. To address this problem, the TCP processing of a guest below the socket interface is performed by the Leulo hypervisor.

Leulo implements a kind of socket outsourcing, which will be discussed in Section 2.3. Our proposed method also performs the TCP processing of a guest in the network stack of the host, which constructs a hypervisor of the guest.

The TCP processing of Leulo has a priority inversion problem. That is, the hypervisor of Leulo executes TCP processing, prior to any guest OSs. In this thesis, we run RT and non-RT guests together and we give a higher priority to the TCP processing of the RT guests and a lower priority to that of the non-RT guests.

2.3 Improving network throughput of virtual machines

2.3.1 Socket outsourcing

Socket outsourcing [27] and similar techniques [29, 57, 72, 73] offload guests' high-level socket operations to the host. These techniques improve throughput by eliminating

message copying and by sending TCP acknowledgment packets efficiently. Section 5.2 will describe details of socket outsourcing.

vPRO [29] offloads the VM's TCP congestion control function to the driver domain or virtual machine that accesses devices. Moreover, it offloads TCP acknowledgment functionality to the driver domain to improve the TCP receive performance. Virtsockets [71] is a socket library that uses shared memory among a host and their guests. The hypervisor copies messages to the guest memory and user applications access them directly.

These systems mainly improve network throughput in virtual machine environments. In this thesis, we archive consistent latency of RT servers as well as high throughput of non-RT servers. We adopt socket outsourcing to improve the throughput of non-RT servers. Because socket outsourcing eliminates message copying, this mitigates the cache pollution caused by the non-RT servers and leads to achieving consistent latency. Furthermore, we eliminate virtual interrupt handling in a guest kernel by extending conventional socket outsourcing.

2.3.2 Accelerating host-guest message passing

XWAY [50] provides a direct communication path between VMs in the same machine. The authors identified that the usual Xen I/O architecture has notable overheads in the inter-domain communications path: 1) TCP/IP processing overheads, 2) page flipping overhead, and 3) long communication path in the same machine. XWAY bypasses domains' TCP/IP stacks by exchanging inter-VM messages via the shared memory inside the machine. By using the shared memory to exchange messages, the communication overhead becomes lower because page flipping is not needed.

Similar as XWAY, XenVMC [79] proposes an inter-VM communication path for the Xen hypervisor. The main difference from XWAY is that XenVMC is implemented below the socket layer and above the transport layer.

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. Our proposed method in this thesis adopts a similar technique and

reduces latency and latency variance.

2.4 Resource reservation and service level objectives

2.4.1 Exclusive physical resources allocation

A data center reserves resources for RT services to avoid Service Level Objective (SLO) violations. As a simple resource reservation, a data center often allocate exclusive physical resources for RT services [16, 18, 91]. This can be achieved, for example, by allocating the threads involved in the processing of messages destined to the RT services to dedicated CPUs [16, 91] or executing an RTOS co-located to a commodity OS in a single machine [31, 66, 95].

With exclusive physical resources allocation, these data centers face the problem of physical resources under-utilization. Several studies show the low average server utilization in most data centers ranging between 7% and 50% [13, 22, 55, 61]. Low utilization not only negatively impacts maintenance and operational costs but also wastes energy resources. Under-utilized physical resources consume about 70% of their peak energy power [70].

In this thesis, we propose a method that reduces the latency and latency variance of RT services without sacrificing CPU utilization. Our proposed method achieves consistent latency for RT servers and at the same time high throughput for non-RT servers within the bound of the consistent latency for RT servers. Furthermore, our method mitigates a cache pollution problem, which is not addressed in those systems allocating exclusive physical resources.

2.4.2 Heracles

Paper [63] proposes Heracles, a dynamic controller that combines hardware and software isolation mechanisms to eliminate Service Level Objective (SLO) violations while maximizing the throughput of non-RT tasks. Heracles implements a resource control algorithm to manage the CPU cores, Last Level Cache (LLC), operating frequency of

processors and network bandwidth. The authors evaluated Heracles with web search, key-value store, and real-time text clustering servers as real-time applications which ran directly in the host OS. They showed experimentally that cache partitioning is important for consistent latency variances in co-located environments.

Both Heracles and this thesis have a similar common goal. That is, reducing the latency and latency variance of RT servers and maximizing the throughput of non-RT servers. While Heracles does not deal with virtual machine environments, this thesis deals with virtual machine environments. Furthermore, unlike Heracles, we propose a software-based method to mitigate the cache pollution problem.

2.4.3 Silo

Guaranteed transmission latency over network links is an essential requirement for keeping SLOs. Paper [45] shows that for predictable latencies, it is necessary guaranteed network bandwidths, guaranteed packet delays, and guaranteed burst allowances. The authors propose Silo, a mechanism that provides guaranteed bandwidth, delay and burst allowances through a VM placement algorithm, coupled with a fine-grained packet pacer. Silo allows datacenter operators to control traffic between VMs in terms of bandwidth, delay, and burst allowance. Silo implements a VM placement algorithm to maximize the number of tenants while meeting network guarantees. Silo has a packet pacer in the network driver of a hypervisor. The modified driver uses a technique that couples I/O batching with dummy packets which are dropped by the first hop switch.

Silo considers the low-level network components, including the physical network switches and the network interface cards (NICs). Silo guarantees network delay, i.e. the delay from the source NIC to the destination NIC. However, Silo does not consider the other network components, including hypervisors and the network stacks of guest OSs. In this theses, we address delays in the hypervisors and the network stacks of guest OSs.

2.4.4 IRMOS/Real-time SOIs

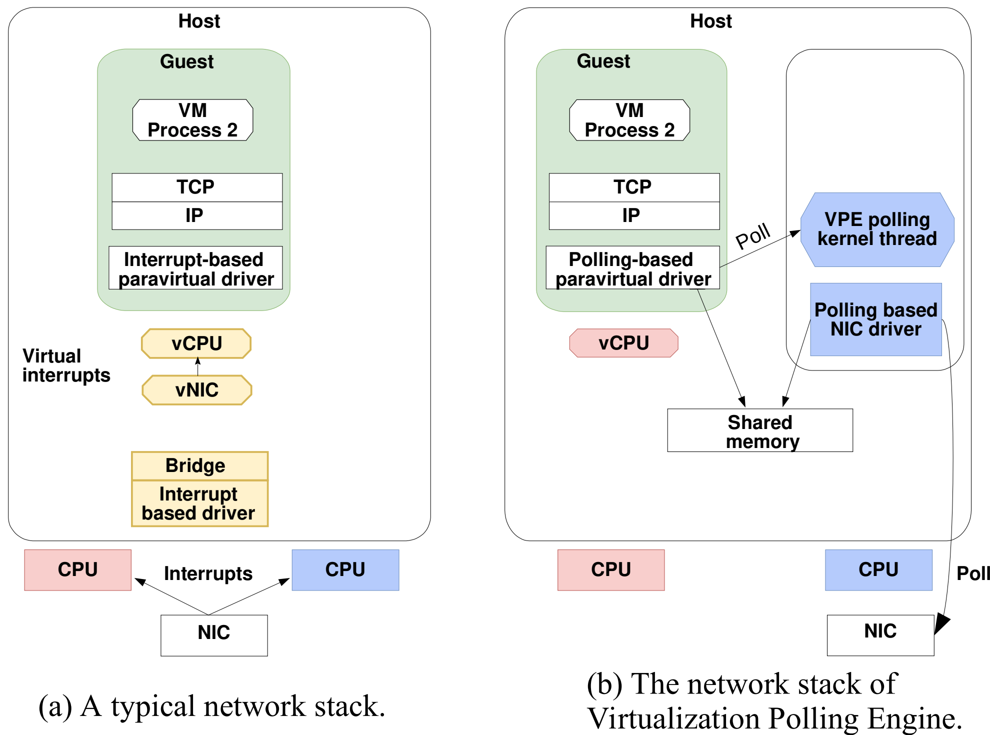
The project of Interactive Real-time Multimedia Applications on Service Oriented Infrastructures (IRMOS) [44,54] proposes an approach for providing real-time Quality of service (QoS) guarantees in Service Oriented Infrastructures (SOIs) at various levels (application, network, storage, and processing). The proposed approach combines several techniques to achieve a predictable execution and to provide QoS support, ranging from the application level to the network resources management level.

While the IRMOS project provides the negotiation and control mechanism for achieving SLO requirement, it does not discuss the implementation of real-time network stacks. They assume that they can achieve SLO requirements by resource reservation. In this thesis, we show that exclusive allocation of the CPU resource, which is a very special case of resource reservation, is not sufficient for implementing real-time network stacks in virtual machine environments. Furthermore, we achieve high throughput of non-RT services, which is achieved by prioritizing without resource reservation.

2.5 Network I/O without interrupt handling

Typical network I/O for VMs suffers from performance degradation. This is caused by many items including context switching among a hypervisor, vCPU threads, I/O backend threads, host interrupt handlers, and so on.

To address this problem, Virtualization Polling Engine (VPE) [62] takes advantage of multi-core processors. VPE uses an entire core for polling a NIC and delivering notifications to VMs through shared memory. Figure 2-7 compares a typical network stack of a hosted virtual machine to the network stack of VPE. Figure 2-7a shows the typical network stack of a hosted virtual machine. In this approach, a VM cannot perform I/O operations to the NIC directly. Instead, each I/O operation requires the intervention of the host network stack, the VM's backend driver (vNIC) and the hypervisor. Figure 2-7b shows the network stack of VPE. The VPE polling thread runs in a dedicated CPU and is never de-scheduled. The thread continuously polls



(a) A typical network stack.

(b) The network stack of Virtualization Polling Engine.

Figure 2-7: Comparison of a typical network stack and that of Virtualization Polling Engine.

the NIC for new messages. VPE has some advantages over the typical network stack. First, the polling thread is never disturbed by the CPU scheduler of the host because it uses a dedicated core. Next, separating the VM threads from the polling thread reduces the number of context switches.

Netmap is a framework that enables userspace applications to send and receive packets by polling mode and eliminates the overheads of interrupt handling in a host OS and guest OSs. In Netmap, all packets are processed by the Netmap framework and the application processes running on top of it. Bypassing the network stacks of OSs offers two main advantages: 1) It eliminates copying messages between the user and kernel memory because messages are copied once from a NIC to the user memory via DMA by the NIC. 2) It eliminates the costs of per-packet dynamic memory allocations in the kernel network stack.

Data Plane Development Kit (DPDK) [59] implements a mechanism similar to Netmap. DPDK has a set of user-level functionalities such as multi-core scheduling

with Non-Uniform Memory Access (NUMA) awareness, and libraries for packet manipulation across cores [7]. Compared to Netmap, DPDK has a disadvantage. When a core of DPDK obtains a NIC, the Linux kernel no longer can use the NIC.

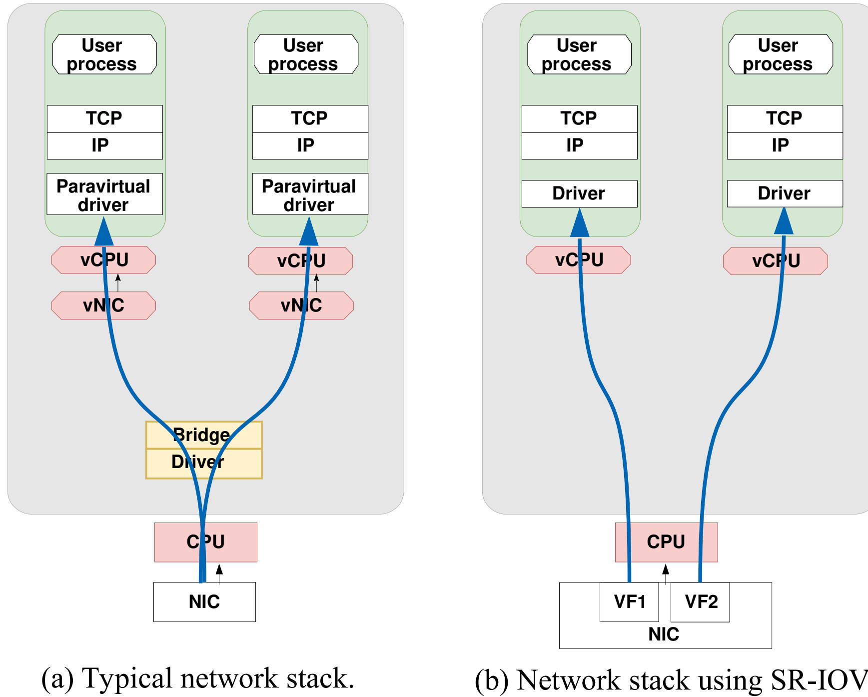
These approaches allow user applications to access network devices without the intervention of the host OS and a guest OS. They avoid message copying by the host OS and reduce the cache pollution. They also avoid interrupt handling, which causes priority inversions in the network stacks of the host and a guest OS. However, they have a significant drawback. That is, user applications must implement their own network stacks using special APIs. In this theses, we achieve consistent latency without modifying user applications. While we modify a guest kernel and add kernel modules to the guest and host kernel, we can run unmodified user applications that use standard socket APIs.

2.6 Network I/O using advanced hardware features

2.6.1 Virtual machine device queues (VMDQs) and Single-root input/output virtualization (SR-IOV)

Interrupts to a VM may interfere with the executions of other co-located VMs. The paper [65] addresses this problem using virtual machine device queues (VMDQs). A VMDQ enabled NIC classifies the receiving packets using the IP addresses or other identifiers of VMs [86]. The paravirtualized device drivers in VMs use shared pages and avoid packet copying between the host and the guests. [20].

Single-root input/output virtualization (SR-IOV) [75] is a specification that allows a physical NIC device to present itself to the OS as multiple separate devices called virtual functions (VFs). Each VF has a dedicated queue for receiving and transmitting packets. Figure 2-8 compares a typical network stack and a network stack using SR-IOV in hosted virtual machine environments. In Figure 2-8a, packets to processes in VMs should go through the network stack in the host kernel to reach the guest kernels. In Figure 2-8b, a NIC with SR-IOV has two VFs for the two VMs. When



(a) Typical network stack.

(b) Network stack using SR-IOV.

Figure 2-8: Comparison of a typical network stacks and a network stack using SR-IOV.

a packet arrives, the NIC checks the IP address or another identifier of the packet, places it via DMA to the memory of the guest OS and injects an interrupt to the guest OS. Using SR-IOV enables bypassing the network stack in the host. Paper [2] identifies that the network stack using SR-IOV has lower mean latency and latency variance than a typical network stack.

In this thesis, we propose a software-based method for consistent latency and compares it with other software-based methods for the same goal. While these hardware-based approaches can achieve consistent latencies, they have a limitation in terms of scalability. For example, the scalability in SR-IOV devices is limited by the number of virtual functions. Paper [38] reports that about the half of the bandwidth is utilized when a single VM runs because the CPU core that executes the VM reaches its maximum utilization. In Chapter 6, through experiments, we show that our proposed method provides high scalability and CPU utilization.

2.6.2 Cache Allocation Technology (CAT) and vCAT

Intel's Cache Allocation Technology (CAT) is a hardware feature for cache partitioning. As discussed in Section 2.4.2, Heracles uses CAT in a non-virtualized execution environment for reducing SLO violations.

Paper [97] proposes vCAT, a CAT manager to achieve hypervisor and VM-level cache allocations. In vCAT, each VM has a number of virtual partitions of LLC, and they are mapped to physical partitions in the hypervisor using a table similar to a page table. Each VM can allocate its virtual partitions to processes dynamically. vCAT considers levels of criticality of the VMs to manage cache partitioning. For example, in real-time VMs, the preemption of physical partitions is disabled.

CAT can avoid cache pollution that is caused by co-located non-RT servers. In this thesis, we propose a software-based method to mitigate the cache pollution problem. This method works in the CPUs that do not have such advanced hardware support. For example, Intel Core i7 processors do not have CAT while Intel Xeon processors have.

2.7 Real-time networks

Links as a Service (LaaS) [98] is an abstraction for a cloud service that provides isolation of network links. In this approach, each VM gets an exclusive set of links and is guaranteed to receive the exact same bandwidth. LaaS implements a scheduler that uses OpenStack [85] for the placement of VMs and configures a Software-Defined Networking (SDN) controller to provide packet forwarding without interference.

QJUMP [37] and PriorityMeister [100] use Deterministic Network Calculus (DNC) [46] to calculate the upper bound of latency and provide bandwidth and latency guarantees [99]. QJUMP prioritizes packets based on flow classes that are set by a network administrator. It uses priorities and rate limiting and allows different traffic classes with different trade-offs between network latency and throughput [45]. A VM of an RT class receives packets with the highest priority and worst-case latency guarantee. A VM of a non-RT class can send packets with higher rates, a lower

priority and no latency guarantee. PriorityMeister mainly focuses on network latency and uses DNC to calculate the worst case latency for each VM based on their rate limits. In contrast to QJUMP, PriorityMeister automatically configures priorities of VMs.

In the same way as QJUMP, Silo [45] uses DNC and guarantees the worst-case packet latency under user-specified rate limits [99]. We have described Silo in Section 2.4.3.

These approaches, LaaS, QJUMP, PriorityMeister, and Silo, consider the low-level network components, including the physical network switches and NICs. They guarantee network delay, i.e. the delay from the source NIC to the destination NIC. However, they do not consider the other network components, including hypervisors and the network stacks of guest OSs. In this thesis, we address delays in the hypervisors and the network stacks of guest OSs over a real-time network. In this real-time network, we assume that the bandwidth and delay are guaranteed as in these approaches.

2.8 Network stacks for high-performance computing

McKernel [31] is an OS designed for high-performance computing (HPC). Similar to RT-Linux, McKernel presents a hybrid design that has a lightweight kernel called McKernel and the Linux kernel. While McKernel isolates the execution of HPC applications, the Linux kernel is leveraged to support the full POSIX API.

The authors propose a framework called Interface for Heterogeneous Kernels (IHK). This framework allows the dynamic partition of systems resources in multi-cores environments. In addition, the framework provides an inter-kernel communication layer.

In IHK/McKernel, the OS of HPC applications provides a system call offloading mechanism to use the functionalities in the Linux kernel. When an HPC application invokes a system call which is not implemented in the Mckernel, the Mckernel sends a system call request to the Linux kernel through the inter-kernel communication layer.

The request is received by a proxy process running on Linux. The proxy process invokes the system call and returns the result back to the HPC application. A unified address space model of IHK/McKernel grants the Linux kernel to access the data and stack of the HPC application when it invokes an offloaded system call.

Our proposed method also implements an offloading mechanism, called socket-outsourcing [27]. However, we use it with a different goal, realizing real-time network stack. By extending socket outsourcing, we eliminate message copying and mitigate the cache pollution problem caused by the non-RT servers.

Chapter 3

Analyzing vanilla Linux and two conventional RT methods

In Chapter 1, we describe our target hosted virtual machine environment of this thesis. This environment hosts RT and non-RT network servers in VMs, as shown in Figure 1-2. Our goal is to achieve short and consistent latency for RT servers and to obtain high throughput for non-RT servers and avoid low CPU utilization within the bound of the consistent latency for RT servers. In this chapter, we identify the causes of latency of RT servers in two conventional RT methods. The first conventional RT method is the *threaded interrupt handling method* which uses the PREEMPT_RT patch to make the kernel more preemptible by translating interrupt handlers into threads. The second conventional RT method is the *the exclusive CPU method* which uses the PREEMPT_RT patch and allocates an exclusive CPU to a group of RT threads. We also analyze the base implementation of Linux called vanilla Linux.

We analyze the network stack of the target environment through preliminary experiments. We measured the latency of an RT server, CPU usage of the host and the throughput of non-RT servers. To find the causes of the problem, we must measure the latencies of individual components of the network stack. However, existing measurement tools were not sufficient because they have non-negligible probe effects. Therefore, we have implemented and used our own tool, called lightweight probes.

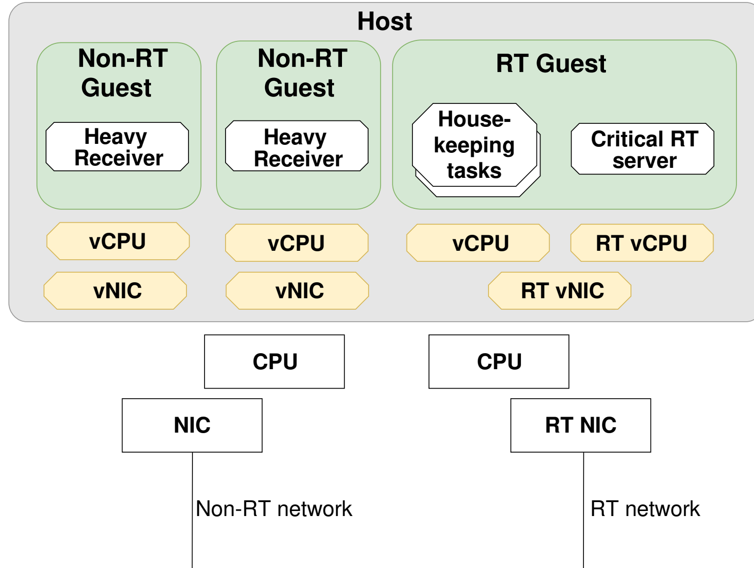


Figure 3-1: Running an RT server and co-located non-RT servers in a target virtual machine environment.

3.1 Experimental environment

In our experiments, we ran two types of servers (Figure 3-1):

- *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 [91]. 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

[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.

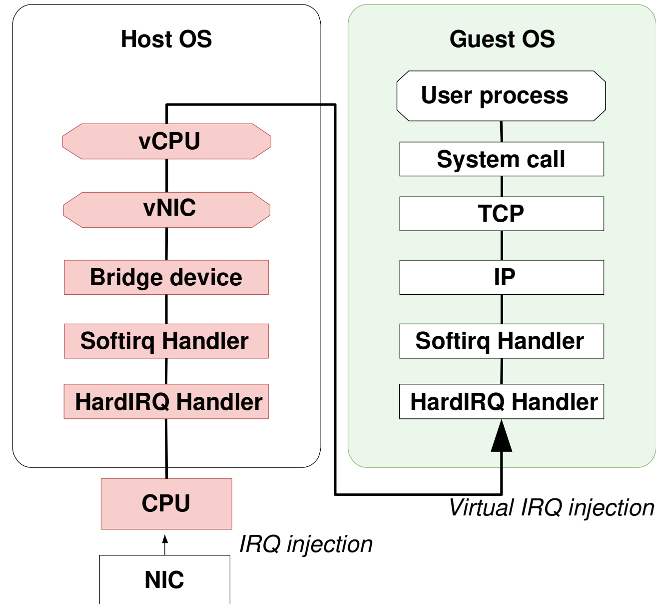


Figure 3-2: The components of the network stack in the target environment.

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 3-1, the host is connected to the following two networks:

- *The RT network*. The bandwidth and delay and are guaranteed as in [15,45,92, 96]. The network interface card (NIC) to this network is labeled an RT NIC.
- *The non-RT network*. This is a best-effort network.

In Figure 3-1, 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.

In all experiments, we used Linux 4.1.10 for the host and guest systems. As a hypervisor, we used the KVM hypervisor which is integrated into the Linux kernel.

Figure 3-2 shows the components of the network stack in the target environment from a NIC to a guest user process. It consists of two main parts: the host OS as a hypervisor and the guest OS. The host OS has the following components: the hard

Interrupt Request (IRQ) handler, softirq handler, the bridge device module, vNIC thread, and vCPU thread. The guest OS has similar components, the hard Interrupt Request (IRQ) handler and softirq handler. The guest OS also has the IP layer, TCP layer, and system call layer. It is not trivial to find the causes of the problem in this complex network stack.

3.2 Vanilla Linux and two conventional RT methods

Figure 3-3 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 [58]. 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 X520 NIC creates multiple hard IRQ and softirq handlers (up to 64) for multiple CPU cores [41]. In Figure 3-3, 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 3-3, 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

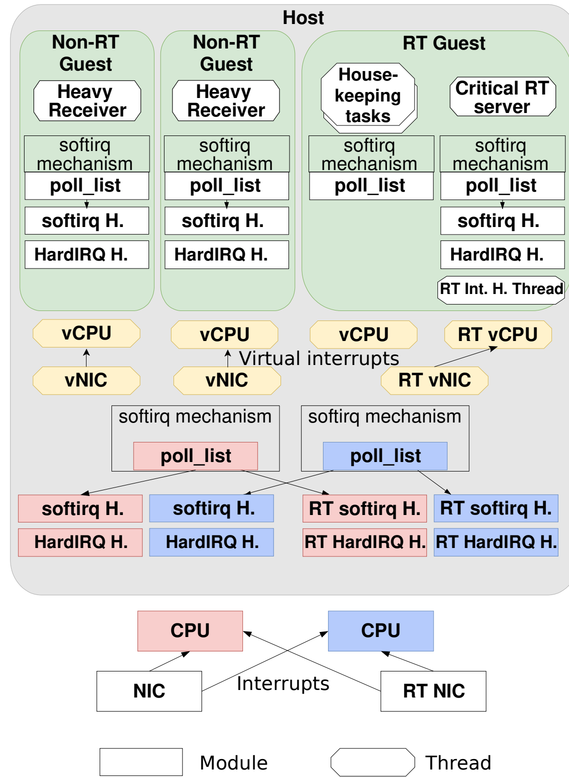


Figure 3-3: Interrupt handling in vanilla Linux.

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 3-3, 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.

It is known that this interrupt handling of vanilla Linux has a priority inversion problem. That is, the kernel of vanilla Linux executes interrupt handlers first, prior to user processes. In Figure 3-3, 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 thesis.

This interrupt handling mechanism of vanilla Linux increases the latency variance

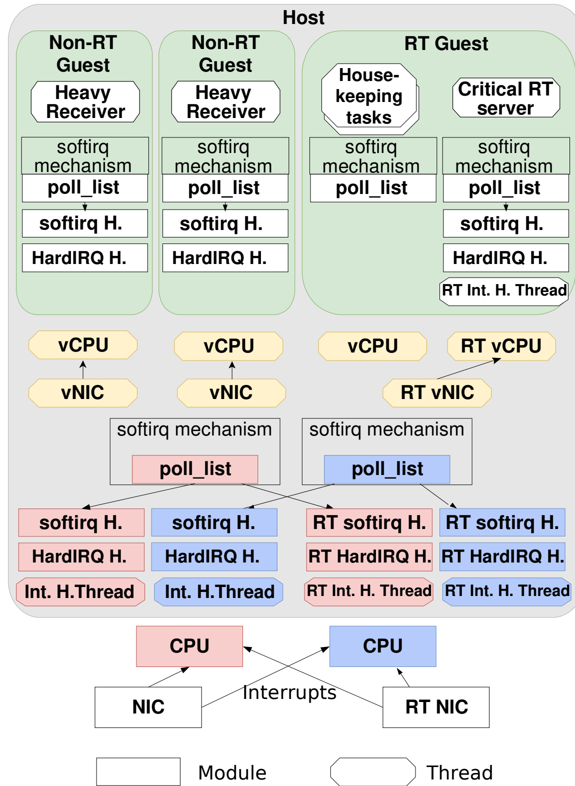


Figure 3-4: Interrupt handling using the threaded interrupt handling method.

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.

To address the priority inversion problem in vanilla Linux, the first conventional RT method uses the PREEMPT_RT patch [84]. We call this conventional RT 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 handler thread, which executes the corresponding hard IRQ and softirq handlers.

- The patch translates spin locks into mutexes that implement a priority inheritance protocol.

Figure 3-4 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 vanilla Linux as follows. Each interrupt handler thread executes the hard IRQ and softirq handlers with its own priority. In Figure 3-4, 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.

While the PREEMPT_RT patch effectively removes the priority inversion in the interrupt-first host kernel of vanilla Linux, this is not sufficient in many data centers. As discussed in Chapter 1 and Section 2.4.1, most existing systems do not solve this problem but bypass the problem. They allocate exclusive physical resources to real-time virtual machines [16, 18, 78]. We call this conventional RT method *Exclusive CPU method*.

Figure 3-5 illustrates interrupt handling in the exclusive CPU method. In this method, we use Linux with the PREEMPT_RT patch and allocate an exclusive CPU (labeled as RT CPU) to a group of threads that executes 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.

Although the exclusive CPU 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

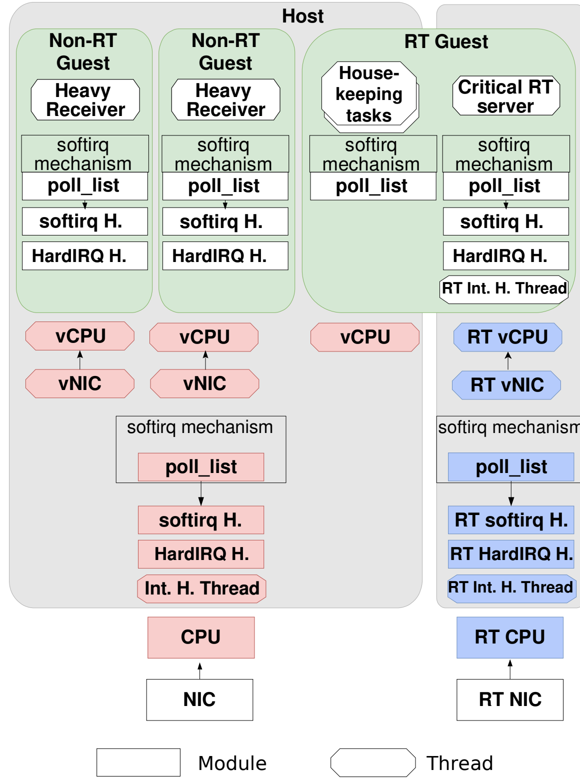


Figure 3-5: Interrupt handling using the exclusive CPU method.

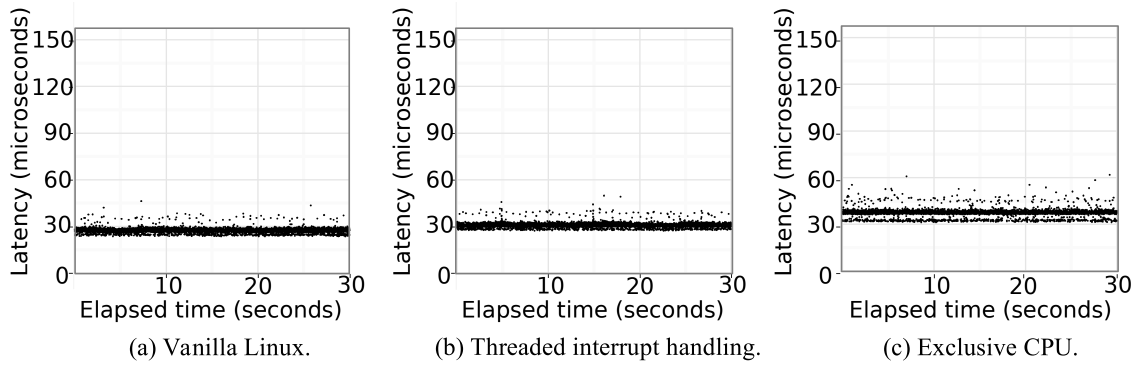
method yields lower CPU utilization.

3.3 Latency and throughput in vanilla Linux and the conventional RT methods

Through a series of experiments, we measured the latency of a Critical RT server and throughput of non-RT servers using vanilla Linux and two conventional RT methods. We describe the details of these experiments in Section 6.1. We activated two CPU cores and used a single RT NIC and two non-RT NICs.

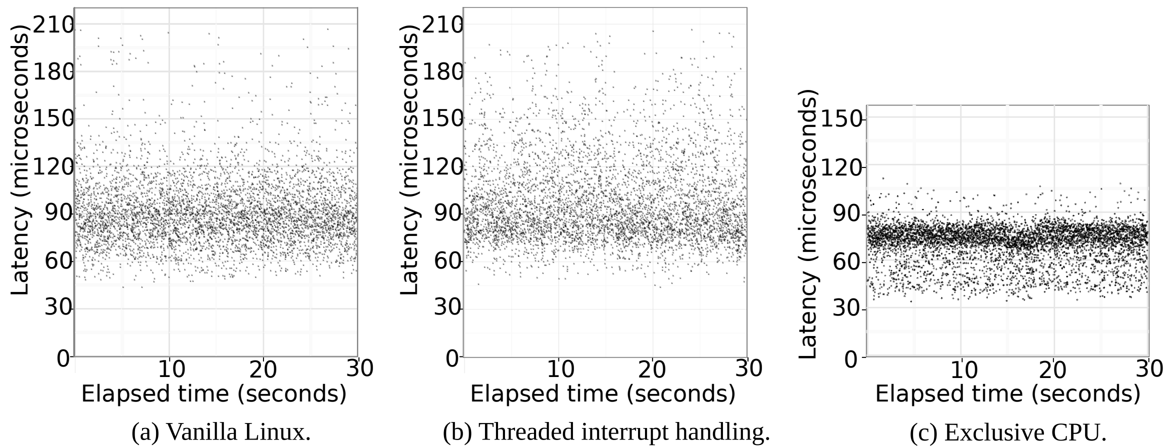
Figure 3-6 and Figure 3-7 show the response times of the Critical RT server co-located without and with two heavy Receiver servers. Table 3.1 and Table 3.2 summarize the statistical values (the mean, 99th percentile, and standard deviation (SD)).

As shown in Figure 3-6 and Table 3.1, without Heavy Receivers, latency and



(a) Vanilla Linux. (b) Threaded interrupt handling. (c) Exclusive CPU.

Figure 3-6: Distribution of the Critical RT server's response times in vanilla Linux and the two conventional RT methods without Heavy Receivers.



(a) Vanilla Linux. (b) Threaded interrupt handling. (c) Exclusive CPU.

Figure 3-7: Distribution of the Critical RT server's response times in vanilla Linux and the two conventional RT methods with Heavy Receivers.

Table 3.1: Statistical values of the Critical RT server response times in vanilla Linux and the two conventional RT methods without Heavy Receivers (microseconds).

Method	Mean	99th percentile	Standard deviation
(non-RT) Vanilla Linux	27.3	34.2	1.5
Threaded interrupt handling	30.7	38.1	1.5
Exclusive CPU	37.4	46.1	2.3

Table 3.2: Statistical values of the Critical RT server response times in vanilla Linux and the two conventional RT methods with Heavy Receivers (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

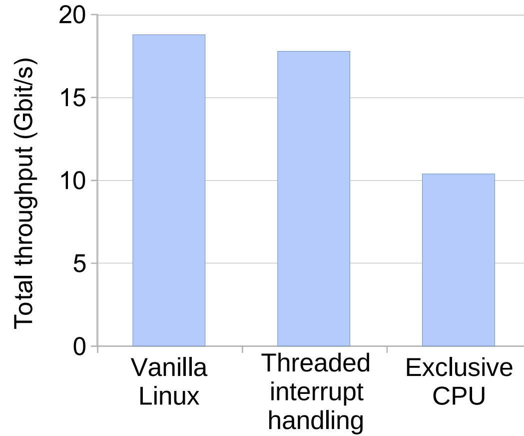


Figure 3-8: Total throughput of Heavy Receivers.

latency variances of the Critical RT server were low in vanilla Linux and two conventional RT methods. As shown in Figure 3.1 and Table 3.2, with two Heavy Receivers, latency and latency variances became larger due to the Heavy Receivers. Among these methods, the exclusive CPU method produced the best latency and latency variance.

At the same time, we measured the total throughput of the Heavy Receivers and the CPU utilization of the VM host. The total throughputs of the Heavy Receivers were shown in Figure 3-8. They were 18.8 Gbps in vanilla Linux, 17.8 Gbps in the threaded interrupt handling method, and 10.4 Gbps in the exclusive CPU method. While the exclusive CPU method produced better latency and latency variance, its throughput was low.

Figure 3-9 shows the CPU utilization of the VM host. As in this figure, 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 more CPU resources than vanilla Linux owing to the overhead of thread context switching. This used the CPU resources more and the throughput was lower than that in vanilla Linux. In Figure 3-9, because only one CPU executed the Heavy Receivers threads and the network stack of the non-RT NICs, the CPU utilization was low (50.8%). This low CPU utilization produced the lowest total throughput.

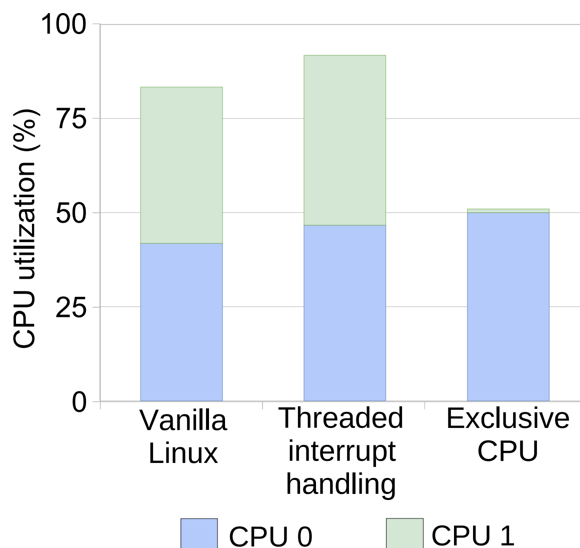


Figure 3-9: Achievable CPU utilization.

3.4 Measuring latencies of network stack components using Ftrace and light-weight probes

In Section 3.3, we have shown that the conventional RT methods had problems. The threaded interrupt handling method had large latency and latency variance. The exclusive CPU method had low latency and latency variance but it sacrificed CPU utilization. However, we did not know the causes of the problem in the threaded interrupt handling method. In this subsection, we analyze the network stack of the target environment in detail and identify the components of the network stack that have large latency and latency variance. Especially, we look into the processing path of messages from the RT NIC to the Critical RT server.

We followed a similar strategy proposed by [34,56] which find the sources of latency variance by dividing a message processing path into smaller segments. We measure latencies of segments using our own tool, called lightweight probes, when existing tools did not work well.

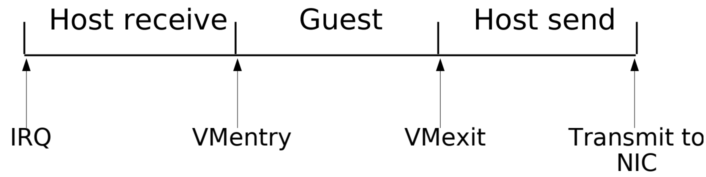


Figure 3-10: Division of the message processing path into three segments.

3.4.1 Processing path analysis with lightweight probes

Before we implemented our own measurement tool, we tried existing tools, such as Ftrace [82] and SystemTap [26]. However, they sometimes did not work well because they have large probe effects. For example, if we enabled Ftrace and activated four trace points, this slowed down the latency around 1.5 μs. This was relatively large in comparison with the latencies of the network stack components.

We have implemented lightweight probes for measuring latencies of components in the target hosted VM environment. Every lightweight probe is identified by a unique number. When a lightweight probe is executed, the probe takes a timestamp 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 a hardware monitor. The probe effect was less than 0.5 μs when using four lightweight probes. Lightweight probes are activated on programmed conditions. For instance, we can get timestamps only when incoming network packets go to an RT server.

3.4.2 Analyzing the message processing path of the Critical RT server with lightweight probes

We divided the message processing path of the critical RT server into the following segments (Figure 3-10):

- **Host receive:** Host execution from the receipt of an IRQ to the start of the guest OS execution (VM entry). This segment includes the network stack pro-

cessing and the execution of the network device backend thread (vNIC thread).

- **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 3.3.

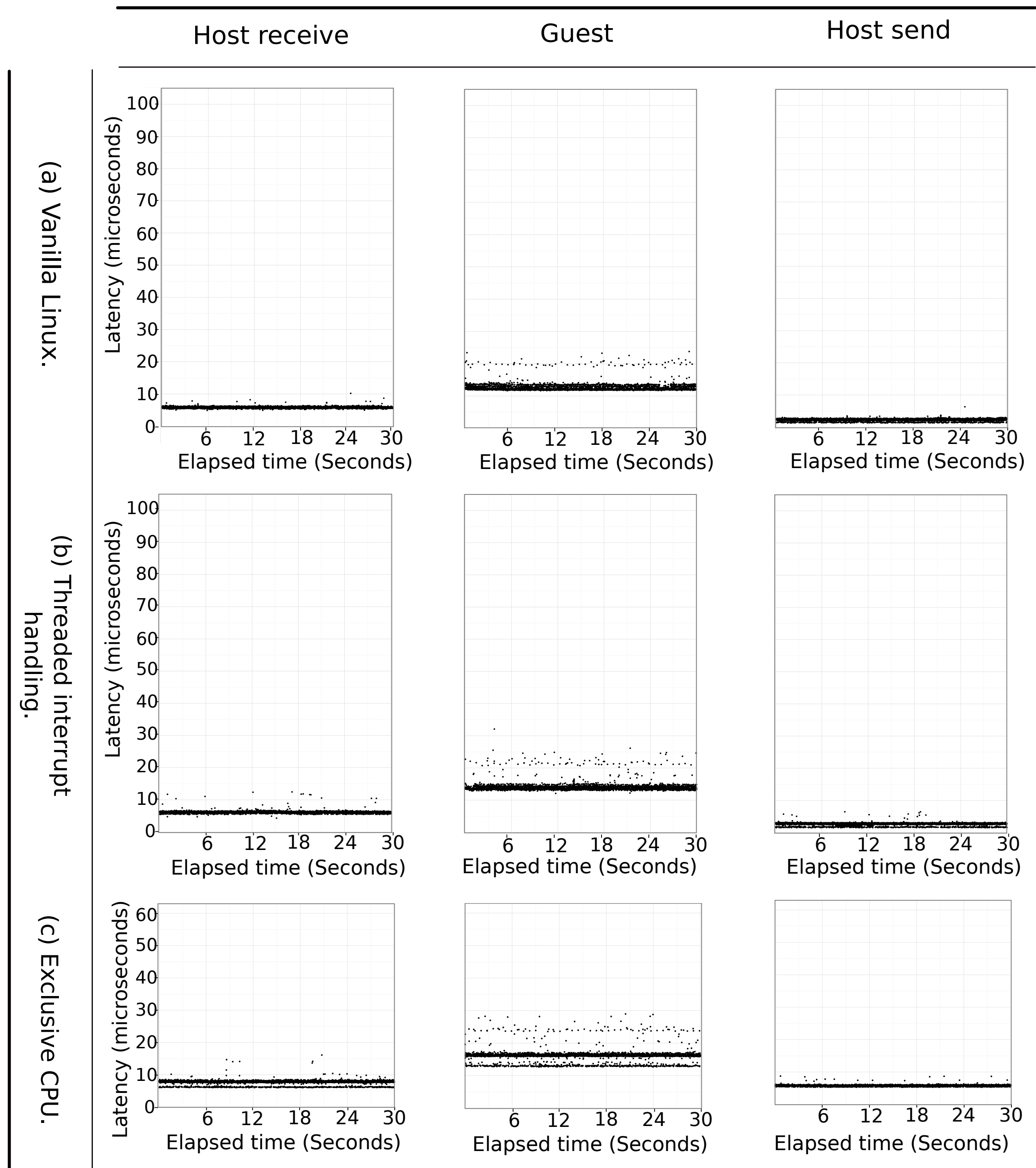


Figure 3-11: Latencies of three segments of the message processing path without running Heavy Receivers.

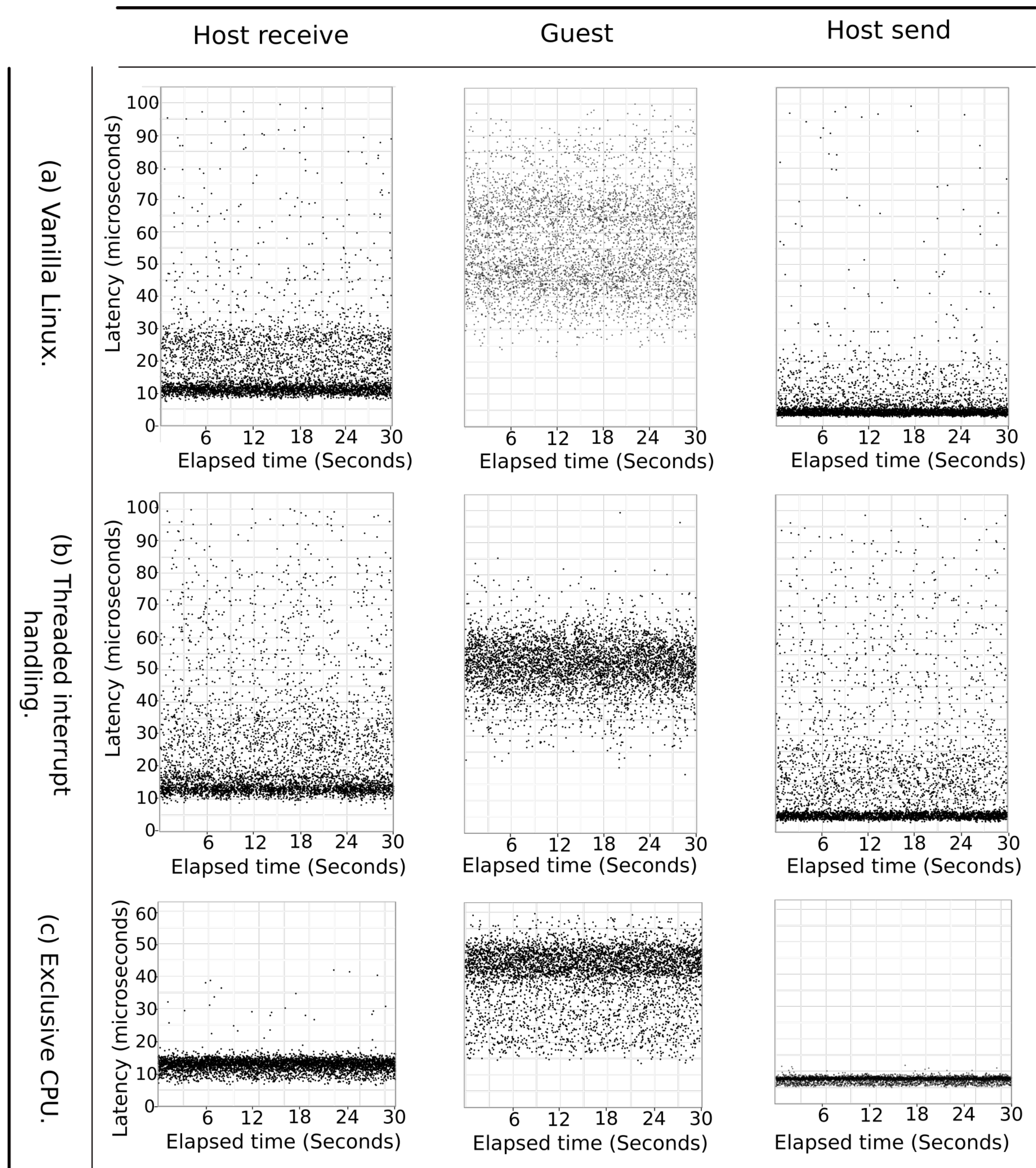


Figure 3-12: Latencies of three segments of the message processing path with running Heavy Receivers.

Figure 3-11 shows the latencies of the segments without running Heavy Receivers. Without running Heavy Receivers, vanilla Linux, the threaded interrupt handling method, and exclusive CPU method had low latencies.

Figure 3-12 shows the latencies of the segments with running Heavy Receivers. Figure 3-12a shows the results using vanilla Linux. By comparing Figure 3-12c, we identified that most of the latency variances were located in the "host receive" segment and the "guest" segment. The results of the threaded interrupt handling method in Figure 3-12b were similar to those of vanilla Linux in Figure 3-12a in this experiment.

Large latency variances in the "host receive" segment were not present when using the exclusive CPU method. In this method, 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 the RT threads of the RT interrupt handler, the RT backend driver, and the RT vCPU.

3.4.3 Finding priority inversions at the "host receive" segment in the threaded interrupt handling method with Ftrace and Kernelshark

We suspected a priority inversion in the "host receive" segment of the threaded interrupt handling method. To identify the culprit functions, we started measuring the execution time of the functions invoked through the segment. We used the Ftrace tool first. As described in Section 3.4.1, Ftrace was not adequate for taking precise measurements. However, with this tool, we identified some functions with high and long tail latencies. For example, we identified that the `net_rx_action()` function was a culprit function. This function is used by the softirq mechanism to process the inbound network traffic. The `net_rx_action()` function executes softirq handlers in the poll_list in a round robin fashion. Next, in the `net_rx_action()` function, we found that the softirq mechanism executes the RT and non-RT softirq handlers in

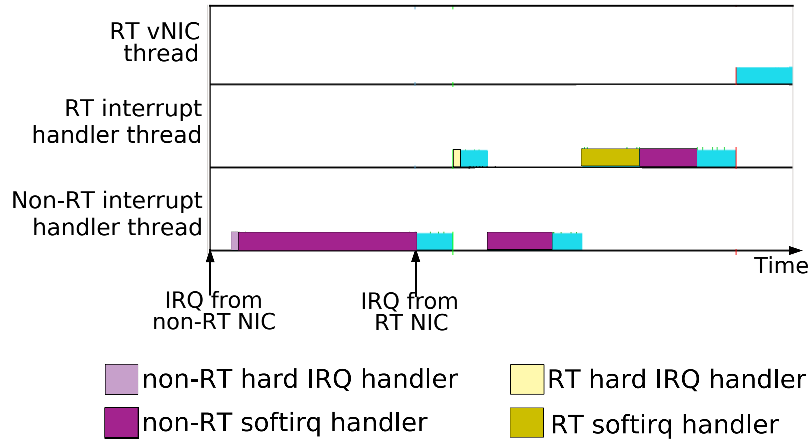


Figure 3-13: Priority inversion in the softirq handling in the host OS using the threaded interrupt handling method.

a fair manner. Next, we inserted the watching points of Ftrace to IRQ and softirq handlers. We used an Ftrace GUI called Kernelshark [83] to visualize the activities.

Figure 3-13 shows a trace of interrupt handling using the threaded interrupt handling method. 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 because this thread had a higher priority than the user process. The interrupt handler thread of the non-RT driver placed the non-RT softirq handler into the CPU's poll_list.

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 3-13, 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 3-13, 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.

3.5 Cache pollution by co-located non-RT Servers

Through the experiments in Section 3.4, we have not found the causes of the large latency and latency variance in the "guest" segment. We also found that the exclusive CPU method also has this problem. Next, we study 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 3-1, we can control the allocation of CPU (cores) using

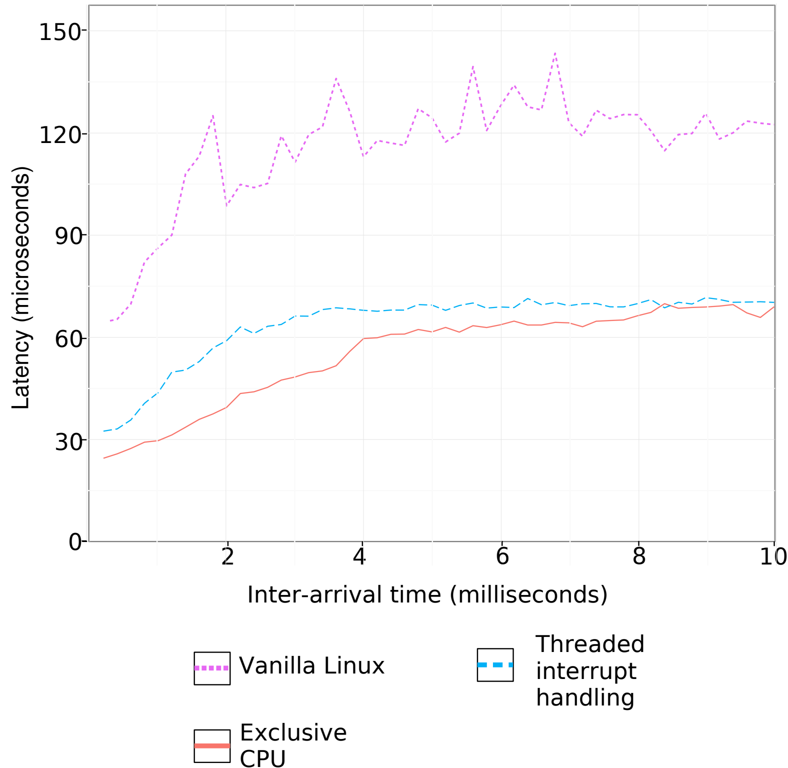


Figure 3-14: 99th percentile latencies of the Critical server in the "guest" segment at different request's inter-arrival times.

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) [40] and ARM's Cache Lockdown [3]. 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 3-4, for example, the Heavy Receivers are receiving a large number of messages persistently from clients. This can pollute the LLC, which can cause latency variance in the Critical RT server.

In this section, we show this cache pollution with an experiment. In this experiment, we ran a single Critical RT server and two Heavy Receivers using vanilla Linux and two RT methods as in Section 3.3. We measured the latency in the "guest" segment with inter-arrival times of requests to the Critical RT server.

Figure 3-14 shows the 99th percentile latencies of the Critical RT server. In Figure 3-14, the x-axis is the inter-arrival time and the y-axis is the latency. As the inter-

arrival time increased, more contents of the Critical RT server were evicted from the cache and the latency became longer. In this experiment, the latency when using the exclusive CPU method had a lower impact than that with the threaded interrupt handling method because this method had a lower throughput. However, the Heavy Receivers interfered with the execution of the Critical RT server because the CPU dedicated to the RT threads shared the LLC with the CPU that executed the other non-RT threads.

In Section 5, 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.

3.6 Summary of analyzing the network stack of a hosted virtual machine environment

This chapter analyzed the network stack of a hosted virtual machine environment through experiments. In these experiments, we ran a Critical RT server and two Heavy receivers in vanilla Linux and two conventional RT methods, the threaded interrupt handling method and the exclusive CPU method. Section 3.1 describes the experimental environment. Section 3.2 describes the interrupt handling in vanilla Linux and two conventional RT methods. Section 3.3 shows the latency of the Critical RT server and the throughput of two Heavy receivers in vanilla Linux and the two conventional RT methods.

Table 3.3 summarizes features of vanilla Linux and these two RT methods. As shown in Section 3.3, the total throughput of the Heavy Receivers was high in vanilla Linux. However, vanilla Linux did not protect the Critical RT server from the Heavy Receivers and had high latency variances. The threaded interrupt handling method also had high latency variance of the Critical RT server. The exclusive CPU method had the lowest latency variances. The total throughput of the Heavy Receivers in the threaded interrupt handling method was high as in vanilla Linux. In the exclusive

Table 3.3: Summary of the RT methods.

Method	Latency variance	CPU util. and total throughput	RT problems and solutions		
			Priority in interrupt	Softirq	Cache
Vanilla Linux	Large	High	Priority inversion	Priority inversion	Cache pollution
Threaded interrupt handling	Large	High	RT Prempt patch	Priority inversion	Cache pollution
Exclusive CPU	Small	Low	RT Prempt patch	Dedicating processors	Cache pollution

CPU method, because RT CPUs did not help to run the Heavy Receivers, this method yielded low CPU utilization and low total throughput of the Heavy Receivers.

In Section 3.4, we divided the message processing path of the Critical RT server into segments and used our own tool, lightweight probes and existing tools Ftrace and KernelShark to measure the latency in the segments. We identified a new priority inversion problem in the softirq handling of the threaded interrupt handling method. The exclusive CPU method bypasses this priority inversion problem by dedicating a CPU to the RT network stack processing and the execution of the RT vCPU.

Finally, Section 3.5 shows that all the methods have the cache pollution problem by the co-located Heavy Receivers.

In this thesis, we do not bypass the priority inversion problem in softirq but solve the problem. Furthermore, we mitigate the cache pollution problem.

Chapter 4

Partitioned RT softirq handling

In Chapter 3, we have described two conventional RT methods, namely, the threaded interrupt handling method and the exclusive CPU method. In this chapter, we describe our proposed method, the "socket outsourcing with RT softirq handling" method or the outsourcing method for short. As shown in this long name, our proposed method consists of two techniques.

- RT socket outsourcing.
- Partitioned RT softirq handling.

In this chapter, we describe the latter technique. In the next chapter, we will describe the former technique.

Figure 4-1 shows the interrupt handling of the outsourcing method. Similar to the threaded interrupt handling method in Figure 3-4, this method avoids the priority inversion problem in the interrupt-first host kernel described in Section 3.2 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 by dividing softirq handling into RT and non-RT types. We will describe this in Section 4.1. In Figure 4-1, these guest kernels have neither the TCP/IP stack nor interrupt handlers. We will describe this in Chapter 5.

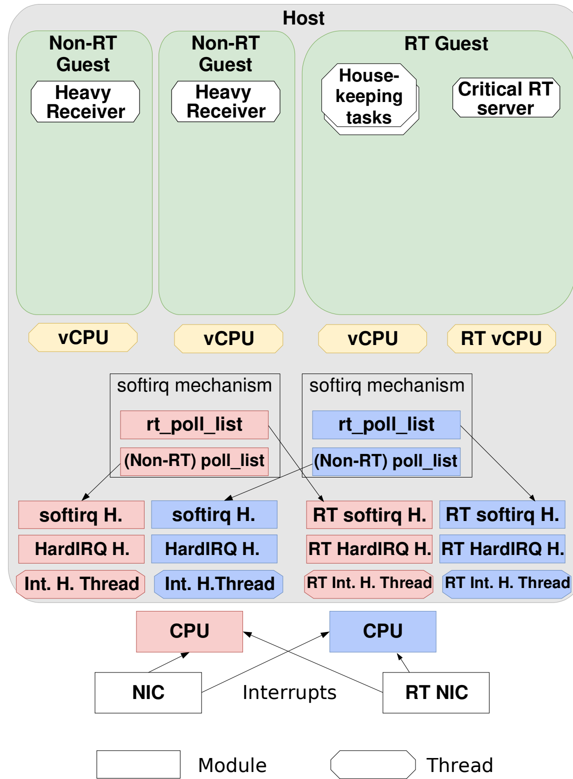


Figure 4-1: Interrupt handling using the outsourcing method.

4.1 Interrupt handling in partitioned RT softirq handling

The outsourcing method divides the poll_list of the softirq mechanism into the following two types (Figure 4-1).

- **(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 4-2 shows a KernelShark trace of interrupt handling using the outsourcing method. As in Figure 3-13 in Section 3.4.3, this CPU received two IRQs. The first was from the non-RT NIC and the second was from the RT NIC.

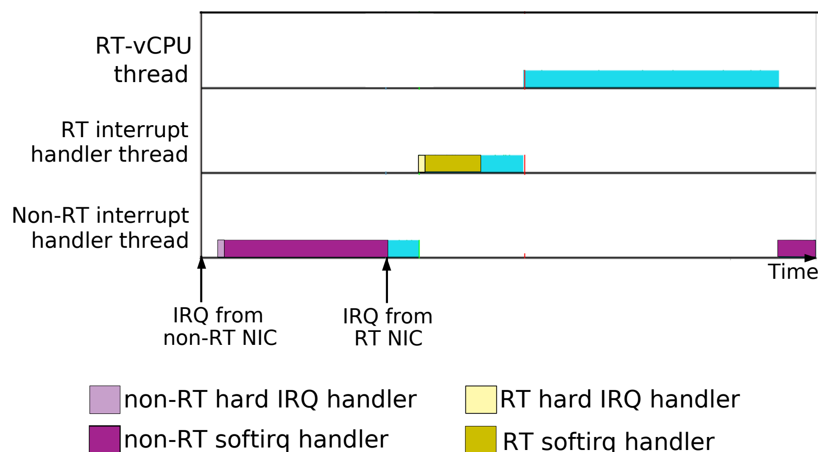


Figure 4-2: The trace of interrupt handling in in partitioned RT softirq handling.

In this figure, the CPU performed the same processing as in the threaded interrupt handling method until the IRQ from the RT NIC arrived. First, 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. 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 4-2, 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 3-13, 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 3-13, 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 vCPU 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 3-13, Figure 4-2 indicates no priority inversion in the softirq handling.

4.2 Implementation of partitioned RT softirq handling

In this section, we explain the implementation of partitioned RT softirq handling.

Linux kernel provides an API of the network subsystem to developers of network device drivers. This API is called the NAPI, which sands for new (network) API [58]. As described in Section 3.2, Linux implements the split interrupt handling model to handle interrupts. Each device driver has two interrupt handlers: the hard Interrupt Request (IRQ) handler and the softirq handler. Softirq handlers are called from the softirq mechanism.

The technique of partitioned RT softirq handling consists of the following two parts:

- Modifying the NAPI module.
- Modifying the softirq mechanism.

These modifications required changing 150 lines of code but did not require changing existing device drivers. These modifications are also 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.

4.2.1 Modifying the NAPI module

First, we have added the rt_poll_list to the NAPI module. The NAPI module has the data structure `softnet_data`, as shown in Figure 4-3. We have added the

```
struct softnet_data {

    struct sk_buff_head process_queue;
    struct list_head    poll_list;
    struct list_head    rt_poll_list;

    /* stats */
    unsigned int                processed;
    unsigned int                time_squeeze;
    unsigned int                cpu_collision;
    unsigned int                received_rps;
...
    unsigned int                dropped;
    struct sk_buff_head input_pkt_queue;
    struct napi_struct  backlog;
    struct sk_buff_head tofree_queue;

};
```

Figure 4-3: Adding rt_poll_list to the softnet_data structure.

```
static struct ctl_table net_core_table[] = {

        {
        .procname       = "rtnet_prio",
        .data           = &sysctl_rtnet_prio,
        .maxlen         = sizeof(int),
        .mode           = 0644,
        .proc_handler   = proc_dointvec_minmax
        },
        ...
}
```

Figure 4-4: Adding sysctl parameter net.core.rtnet_prio.

rt_poll_list to the `softnet_data` (Line 5 of Figure 4-3) as similar to the poll_list (Line 4 of Figure 4-3). Both the poll_list and the rt_poll_list are lists of softirq handlers. The structure `list_head` implements a generic list in the Linux kernel.

Next, 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-wnet.core.rtnet_prio=47`, interrupt handler threads with a priority higher or equal to 47 use the rt_poll_list.

Figure 4-4 shows the implementation of the sysctl parameter `net.core.rtnet_prio`. We have added the Lines 4 to 10 to the structure `ctl_tablenet_core_table`, which represents the sysctl parameter under `net.core`. The function `proc_dointvec_minmax()` at Line 9 stores the sysctl parameter `net.core.rtnet_prio` to the global variable `sysctl_rtnet_prio` at Line 6.

```
void napi_schedule_irqoff(struct napi_struct *n)
{
        unsigned long flags;

        local_irq_save(flags);
        if(task_prio(current) >= sysctl_rtnet_prio)
          ____napi_rt_schedule(this_cpu_ptr(&softnet_data), n);
        else
          ____napi_schedule(this_cpu_ptr(&softnet_data), n);
        local_irq_restore(flags);
        preempt_check_resched_rt();
}
EXPORT_SYMBOL(napi_schedule_irqoff);

static inline void ____napi_rt_schedule(struct softnet_data *sd,
                                        struct napi_struct *napi)
{
        list_add_tail(&napi->poll_list, &sd->rt_poll_list);
        __raise_rt_softirq_irqoff(RT_NET_RX_SOFTIRQ);
}
```

Figure 4-5: Modifying the function napi_schedule_irqoff().

The NAPI module exposes a number of functions. Finally, we have modified the function `napi_schedule_irqoff()` of these NAPI functions. This function is called from the hard IRQ handler of a NIC driver. This function takes the parameter `n`, which points to the softirq handler of the NIC driver.

In Figure 4-5, the function `napi_schedule_irqoff()` checks the priority of the current thread, which is an IRQ handler thread. If the priority is equal to or greater than the value of `sysctl_rtnet_prio`, the function calls `____napi_rt_schedule()`. The function `____napi_rt_schedule()` puts the softirq handler `n` to the rt_poll_list, as shown in Line 15 of Figure 4-5. Next, function `____napi_rt_schedule()` raises a new softirq, called `RT_NET_RX_SOFTIRQ`. We will describe this in Section 4.2.2. If the priority is less than the value of `sysctl_rtnet_prio`, the function `napi_schedule_irqoff()` calls `____napi_schedule()`. The function `____napi_schedule()` puts the softirq handler `n` to the non-RT poll_list.

Because we did not change the interface of `napi_schedule_irqoff()`, we can reuse the existing device drivers of NICs without any changes. For example, we did not change any code of the device driver of the Intel X520 NIC, which is used in all the experiments in Chapter 3 and Chapter 6.

```
enum
{
        HI_SOFTIRQ=0,
        TIMER_SOFTIRQ,
        NET_TX_SOFTIRQ,
        RT_NET_RX_SOFTIRQ,
        NET_RX_SOFTIRQ,
        BLOCK_SOFTIRQ,
        IRQ_POLL_SOFTIRQ,
        TASKLET_SOFTIRQ,
        SCHED_SOFTIRQ,
        HRTIMER_SOFTIRQ,
        RCU_SOFTIRQ,

        NR_SOFTIRQS
};
```

Figure 4-6: Adding a new softirq kind “RT_NET_RX_SOFTIRQ”

4.2.2 Modifying the softirq mechanism

The softirq mechanism of Linux mimics a hardware interrupt controller and calls the softirq handlers of device drivers. Vanilla Linux has 11 types of softirqs, including `NET_TX_SOFTIRQ` for sending network messages and `NET_RX_SOFTIRQ` for receiving network messages. A hard IRQ handler sets a bit of a bitmap in the per-CPU variables of the CPU that receives the IRQ. For example, the hard IRQ handler of a block device driver sets the bit of `BLOCK_SOFTIRQ`. The softirq mechanism checks the bitmap after all hard IRQ handlers finish. If a bit is set, the softirq mechanism calls the corresponding softirq manager. For example, the softirq mechanism calls the function `net_rx_action()` for the bit of `NET_RX_SOFTIRQ`. At this time, the softirq mechanism acquires the softirq_lock for `NET_RX_SOFTIRQ` in the per-CPU variables. The function `net_rx_action()` processes the poll_list of the structure softnet_data in the per-CPU variables. This poll_list includes softirq handlers of network devices that have pending incoming messages.

We have added a new softirq type, `RT_NET_RX_SOFTIRQ`, at Line 6 of Figure 4-6. The bit of `RT_NET_RX_SOFTIRQ` is set by the function `____napi_rt_schedule()` as described in Section 4.2.1. We have also added the rt_softirq_lock for `RT_NET_RX_SOFTIRQ` as similar to the (non-RT) softirq_lock for non-RT `NET_RX_SOFTIRQ`.

Next, we have copied the `net_rx_action()` and made the new function `rt_net_`

`rx_action()`. This function processes the rt_poll_list of the structure softnet_data in the per-CPU variables.

Chapter 5

RT socket outsourcing

In this thesis, we describe our proposed method, the "socket outsourcing with partitioned RT softirq handling" method or the outsourcing method for short. As shown in this long name, our proposed method consists of two techniques.

- RT socket outsourcing.
- Partitioned RT softirq handling.

We have described the latter method in Chapter 4. In this chapter, we describe the former technique. This technique mitigates the cache pollution problem described in Section 3.5 and avoids the priority inversion problem in a guest's softirq handling by extending conventional socket outsourcing [27].

The following subsections are structured as follows. Section 5.1 describes the conventional socket outsourcing. Section 5.2 describes our proposed technique, RT socket outsourcing. Finally, Section 5.3 shows the implementation details of RT socket outsourcing in Linux KVM.

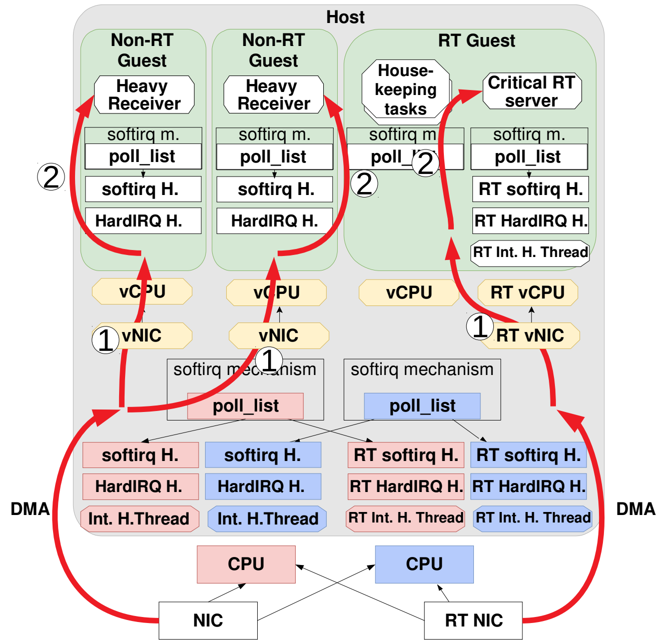
5.1 Conventional socket outsourcing

To overcome the cache pollution problem and the priority inversion problem in a guest, we extend socket outsourcing [27]. 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. The implementation of socket outsourcing uses Virtual Machine Remote Procedure Call (VMRPC) [27] 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 a guest and the host.

Similar to the 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 procedure `recvfrom()` returns the received message to the guest client.

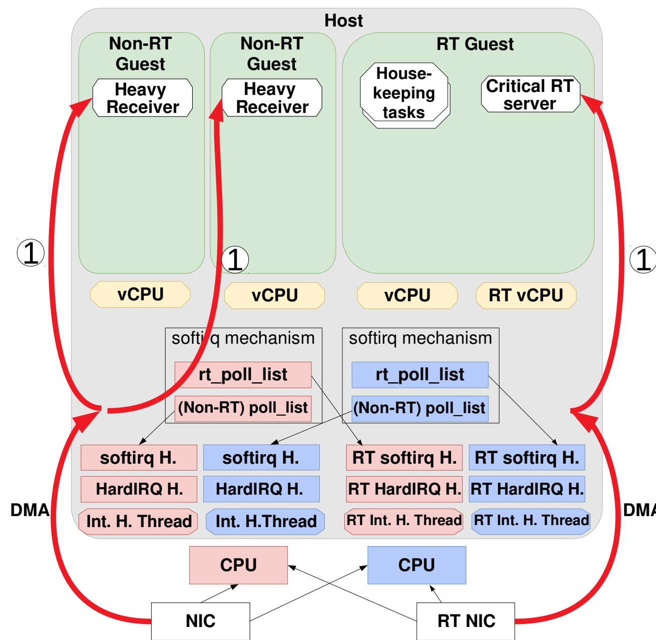
The current production RT methods, i.e., the threaded interrupt handling method and the exclusive CPU method, perform message copying two times. One occurs from the host kernel to a guest kernel, and the other occurs from the guest kernel to a guest user process. In contrast, socket outsourcing requires 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.



① Copy from the host kernel to the guest kernel.

② Copy from the guest kernel to the guest user process.

(a) Threaded interrupt handling method.



① Copy from the host kernel to the guest kernel.

(b) Conventional socket outsourcing.

Figure 5-1: Comparison of network paravirtualization in the threaded interrupt handling method and conventional socket outsourcing.

Figure 5-1 compares the threaded interrupt handling method and socket outsourcing. In Figure 5-1a, message copying is done twice. One occurs from the host kernel to a guest kernel, and the other occurs from the guest kernel to a guest user process. When a message arrives to a guest, the vNIC thread copies the message from the host kernel to the guest kernel memory. Next, the network stack of the guest copies the message from the guest kernel to the socket's buffer of the process. This duplicated copying pollutes the LLC, which causes latency variance in the Critical RT server. The exclusive CPU method has the same problem.

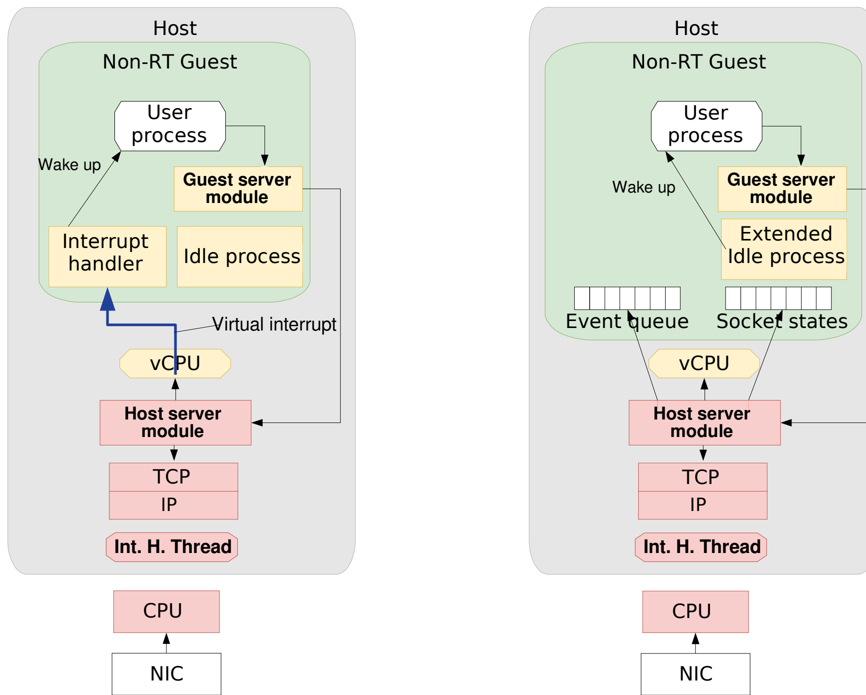
In socket outsourcing, message copying solely occurs from the host kernel to a guest user process. In Figure 5-1b, the guest user processes issue receiving socket-level system calls (such as `read()`, `recv()` and `recvfrom()`), and the guest kernels delegate their processing to the host. When a message arrives to the host, the network stack of the host receives the message from a NIC and passes it to the guest process directly.

5.2 RT socket outsourcing

Conventional socket outsourcing can face the priority inversion problem in the softirq mechanism as described in Section 3.2 because it makes use of virtual interrupts as described in Section 5.1. We solve 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* of a guest OS. 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 up. This



(a) Conventional socket outsourcing.

(b) RT socket outsourcing.

Figure 5-2: Comparison between conventional socket outsourcing and RT socket outsourcing.

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.

Figure 5-2 compares conventional socket outsourcing to RT socket outsourcing. In this figure, both the user processes are waiting for a message and they issue a receive system call. The kernel thread of each user process executes its guest client module of VMRPC, and performs a VMRPC to the host server module. Each host server module returns nothing soon because the host kernel has no message. Each vCPU thread is sleeping.

In conventional socket outsourcing in Figure 5-2a, when a message arrives, the host server module of VMRPC notices about it and injects a virtual interrupt to the

guest kernel. The interrupt handler in the guest kernel processes wakes up the kernel thread of the guest user process. The kernel thread of the user process executes the guest client module of VMRPC, and performs a VMRPC to the host server module. The host server module copies the message to the guest user process because the host kernel has the message. Finally, the host server module returns the result to the guest client module.

In RT socket outsourcing in Figure 5-2b, when a message arrives, the host server module of VMRPC notices about it and puts an event to the event queue in the guest kernel and changes the state of the receiving socket. Next, the host server module wakes up the vCPU thread. The vCPU threads enters the virtual machine and executes the extended idle process. The extended idle process wakes up the kernel thread of the user process without interrupt handling. The following process is the same as that in the conventional socket outsourcing.

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 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. We will compare, the latency of this RT socket outsourcing with that of conventional interrupt-based socket outsourcing in Section 6.3.

5.3 Implementation details of RT 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.

5.3.1 The guest client module

We load a module to a guest kernel. This module acts as a client of VMRPC. This module modifies the system call table and overrides the functions of sockets. In this section, we describe the implementation of the guest client module using the function of the system call recvfrom() as an example.

Figure 5-3 shows the function p_recvfrom(), which implements the system call recvfrom(). This function takes the following arguments.

- sockfd: Specifies the socket file descriptor.
- buf: Points to the buffer where the message should be stored.
- len: Specifies the length in bytes of the buffer.
- flags: Specifies the type of message reception.
- src_addr: A null pointer, or points to a sockaddr structure in which the address of a sender is stored.
- addrlen: Specifies the length of the sockaddr structure.

This function returns the length of the received message in bytes. If no messages are available, this functions returns zero. If an error occurs, this function returns a minus value of an error code according to the Linux system call convention.

```
1  asmlinkage long p_recvfrom(int sockfd, void *buf, size_t len, int flags,
2                         struct sockaddr *src_addr, int *addrlen)
3  {
4      struct socket *sock;
5      struct sock *sk;
6      int err=0;
7      int hfd;
8      unsigned int res;
9      int fput_needed;
10     int mask;
11
12
13     sock = sockfd_lookup_light(sockfd, &err, &fput_needed);
14     if (!sock)
15           return err;
16
17     sk = sock->sk;
18     lock_sock(sk);
19     hfd = sk->sockfd;
20
21 b_loop: //waiting for socket events
22
23     mask = p_poll_sock_(sk);
24     if(mask == POLLERR)
25             goto addrrunlock;
26
27     if(!(mask & (POLLIN | POLLPRI)))
28     {
29       long timeo = sock_rcvtimeo(sk, flags & O_NONBLOCK);
30       err = _wait_for_mask(sk, timeo, POLLIN | POLLPRI);
31
32       if (err)
33         goto addrrunlock;
34
35       goto b_loop;
36     }
37
38     res =  sguest_recvfrom(hfd, buf, len, flags, src_addr, addrlen);
39
40     err=res;
41
42 addrrunlock:
43
44     fput_light(sock->file, fput_needed);
45     release_sock(sk);
46     return err;
47 }
```

Figure 5-3: Implementation of the `recvfrom()` system call in a guest.

First, this function translates the file descriptor sockfd in the guest into the pointer to struct socket with sockfd_lookup_light(). Next, this function obtains `sk`, the pointer to another structure struct sock. Next, this function checks if the message is available with the function p_poll_sock_(), which accesses the socket states in the shared memory between the host and the guest. If there is no message, the function calls the function _wait_for_mask() and makes the current thread of the user process

sleep mode in the guest. If there is a message, the function performs a VMRPC with the function sguest_recvfrom() to the host server module. The arguments of this VMRPC are the same as the function p_recvfrom() except for the file descriptor `hfd`. The file descriptor `hfd` is a file descriptor in the host kernel. The function p_recvfrom() returns the same value as the VMRPC.

5.3.2 The Host server module

We load a module to a host kernel. This module acts as a server of VMRPC. In this section, we describe the implementation of the host server module using the function of the system call recvfrom() as an example.

Figure 5-4 shows the function skhst_recvfrom(), which implements the server procedure of the client p_recvfrom() in Section 5.3.1. First, this function obtains the arguments from p_recvfrom() with the function vmrpc_copy_from_guest(). The arguments are the same arguments as those of the guest client p_recvfrom() except for the file descriptor. The file descriptor is a file descriptor in the host kernel. The pointers (buf, addrlen and src_addr) are also in the guest logical address.

Next, this function translates the pointers `buf`, `addrlen`, `src_addr` in the guest logical address into those in the host logical addresses. The function calls the function sys_recvfrom(), which implements the system call recvfrom() in the host kernel. Next, this function updates socket states in the shared memory with the function update_mask(). The function returns the same value as sys_recvfrom() to the guest client module.

Figure 5-5 shows the main code of the function notify_guest(). This function takes the pointer to a struct sock as an argument. This function is called when the host network stack notices the change of the status of the given socket. For example, when a new message arrives to a socket which has no message before, this function is called.

First, the function obtains the identifiers of the vCPU (vcpu_id and vcpu). Next, the function copies the status of the socket in the mask variable. Next, the function calls _create_event() which updates the status of the socket in the memory of the

```
static int skhst_recvfrom(void *rpcdata,gva_t arg_gp,size_t bytes)
{

        struct {int sockfd; gva_t buf; int len; int flags;
                        gva_t src_addr; gva_t addrlen;} data;


        void * buff;
        rpck_data *kdata;
        kdata = rpcdata;
        struct sockaddr *src_addr;
        int *addrlen;
        unsigned int size;


        if(bytes != sizeof(data))
        {
                return -EFAULT;
        }
        if(vmrpck_copy_from_guest(rpcdata,&data,arg_gp,bytes)) return -EFAULT;

        if(!data.buf)
                return -EINVAL;

        buff =(void*)skhst_get_hva(rpcdata,(gva_t)data.buf);

        if(data.src_addr)
        {
                src_addr = (struct sockaddr *) skhst_get_hva(rpcdata,
                                                        (gva_t)data.src_addr);
                addrlen = (int *) skhst_get_hva(rpcdata,(gva_t)data.addrlen);
        }


        size = sys_recvfrom(data.sockfd, buff, data.len,data.flags, src_addr,
                        addrlen);
        update_mask(data.sockfd, 0);

        return size;
}
```

Figure 5-4: Implementation of the `skhst_recvfrom()` function in the host.

vCPU and puts an event to the event queue of the memory of the vCPU if the status of the socket has changed. Finally, the function wakes up the vCPU thread with swait_wake_interruptible().

5.3.3 The extended idle process

The idle process of Linux is a per-CPU process that runs whenever there is no other runnable process on that CPU. The idle process performs some sanity checks and executes the halt instruction. In a physical processor, the halt instruction halts a

```
1
2 static struct pid __rcu * notify_guest(struct sock *sk)
3 {
4     struct socket *sock;
5     int mask;
6     int vcpu_id;
7     struct kvm_vcpu *vcpu;
8
9     mask = -1;
10
11    vcpu_id = _get_vcpu_id(sk);
12    vcpu = _get_vcpu_by_id(sk, vcpu_id);
13
14    if(!vcpu)
15          return NULL;
16
17    sock = sk->sk_socket;
18    mask = sock->ops->poll(sock->file,sock,NULL);
19
20    _create_event(mask, sk);
21
22    /*wake up the vcpu thread*/
23    if(swaitqueue_active(&vcpu->wq))
24    {
25        swait_wake_interruptible(&vcpu->wq);
26    }
27
28    if(vcpu)
29      return vcpu->pid;
30 }
```

Figure 5-5: Implementation of the `notify_guest()` function in the host.

CPU core until an external interrupt is triggered. In a vCPU, the halt instruction causes the hypervisor to take control of the vCPU thread [42]. The vCPU thread sleeps in the host kernel until an external event is triggered.

We made minimal changes to the idle process of Linux. Figure 5-6 shows the function simple_cpu_idle(), which is the main loop of the idle process of Linux. At Line 22 of Figure 5-6, the idle process calls the function cpuidle_idle_call() which executes the halt instruction. We inserted calling vmrpc_pending_notifications() at Line 5 of Figure 5-6. This function vmrpc_pending_notifications() checks the event queue and wakes up user processes, as described in Section 5.2, and returns a false when the queue has some event(s). We did not change the other lines.

```
void simple_cpu_idle(void)
{

        while (!need_resched() &&
              vmrpc_pending_notifications()) {

                check_pgt_cache();
                rmb();

                local_irq_disable();
                arch_cpu_idle_enter();


                if (cpu_idle_force_poll || tick_check_broadcast_expired())
                {
                        cpu_idle_poll();
                        arch_cpu_idle_exit();
                }

                else
                {
                        cpuidle_idle_call();
                        arch_cpu_idle_exit();
                }

        }
}
```

Figure 5-6: The extended idle process

Chapter 6

Experimental evaluation

In Chapter 4 and Chapter 5, we have described our proposed method, the outsourcing method. In this Chapter, we evaluate the outsourcing method by comparing it with the two conventional RT methods. First, we repeated the experiments in Chapter 3 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 the same chapter. In these experiments, we measured the total latency of the Critical RT server, and latencies of the components in the message processing of the Critical RT server. Next, we evaluate the outsourcing method using application benchmarks. We ran a Voice-over-IP (VoIP) server and a key-value store server as an RT server. Next, we evaluate scalability of the outsourcing method in the number of RT VMs. Finally, we discuss the current limitations of the outsourcing method.

6.1 Experimental setup for running a simple RT server

We have performed experiments using a simple RT server in the experimental environment shown in Figure 6-1. First, we ran netperf [47] 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 [89] 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.

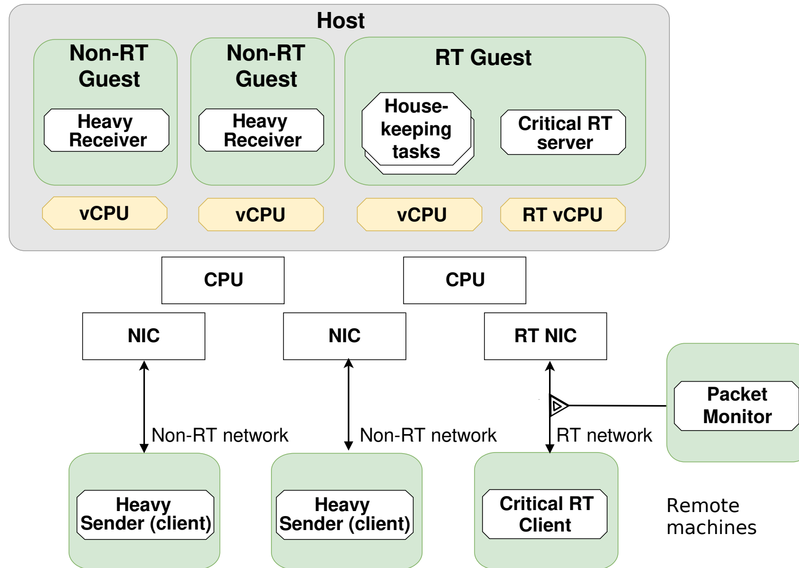


Figure 6-1: The experimental environment.

We emphasize that varying inter-arrival times caused a similar impact to the LLC as varying the load of non-RT Heavy Receivers. 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 load of non-RT Heavy Receivers 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 6-1, 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 X520 Ethernet converged network adapters as the NICs [41]. We connected the VM host to two

Table 6.1: 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

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 the hardware monitor, Endace DAG 10X2-S card [28]. We chose to use the hardware monitor because it had no probe effect. The RT network in Figure 6-1 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 6.1. 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 3.2. In other methods including vanilla Linux, both CPUs ran any threads. We activated all the cores of the other machines. We also performed experiments using the threaded interrupt handling method and executing the Critical RT server and Heavy Receivers directly in the host. We call this execution environment *a non-virtualized environment* environment. The OSs running on the physical machine were Linux 4.1 except for the packet monitor. The machine for the packet monitor ran on Linux 3.16. The version of all guest OSs was Linux 4.1.

Table 6.2: 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

To eliminate fluctuations in the hardware, we turned off the following hardware features: Hyper-Threading, TurboBoost, and C-States[1]. 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 6.2 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 4.2.1, we set `sysctl -w net.core.rtnet_prio=47` and made the RT IRQ handler use the rt_poll_list.

6.2 Experimental results using a simple RT server

We ran the simple RT server as in Section 6.1 for 30 s using the following methods:

- Non-virtualized environment (the host).
- (non-RT) Vanilla Linux.
- The threaded interrupt handling method.
- The exclusive CPU method.
- The outsourcing method.

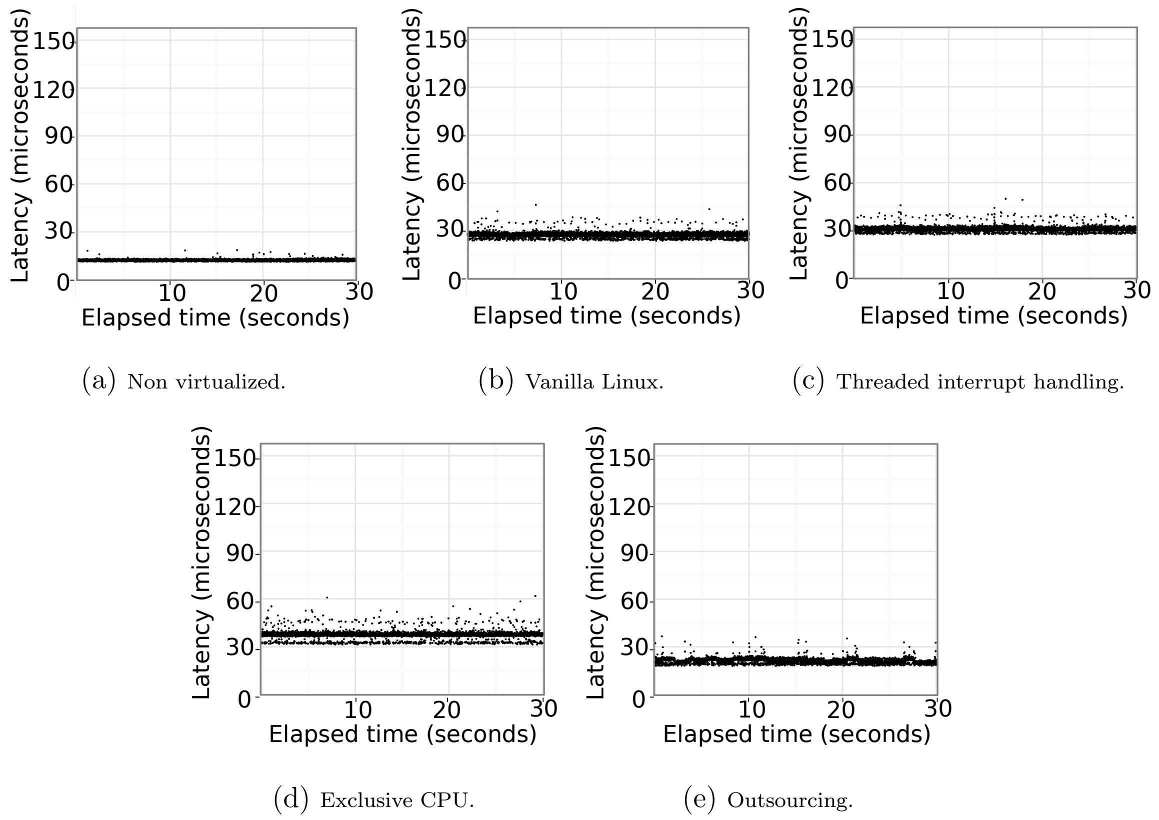
[1]C-states are CPU modes for saving power. C-state transitions degrade the performance of RT servers. We turned o C-states in the BIOS and in the Linux kernel using the parameters intel_idle.max_cstate=0 and idle=poll.

Figure 6-2 presents the experimental results of the Critical RT server without running the Heavy receivers. Table 6.3 summarize the statistical values (the mean, 99th percentile, and standard deviation (SD)). The experimental results were obtained with the hardware monitor as described in Section 6.1. In all the methods, the response times of the Critical RT server had low latency variance. In the outsourcing method, the mean was lower than those in vanilla Linux and the conventional RT methods because the message processing path is shorter.

Figure 6-3 and Table 6.4 present the experimental results of the Critical RT server with running two Heavy Receivers. 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 6-4 and Figure 6-5, respectively.

The outsourcing method produced the lowest latency and latency variance among three RT methods, as shown in Figure 6-3. As shown in Table 6.4, 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 and slightly higher CPU utilization by 5.8% compared to the non-virtualized environment. Compared to vanilla Linux, the outsourcing method produced the same high throughput of 18.8 Gbps.

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.



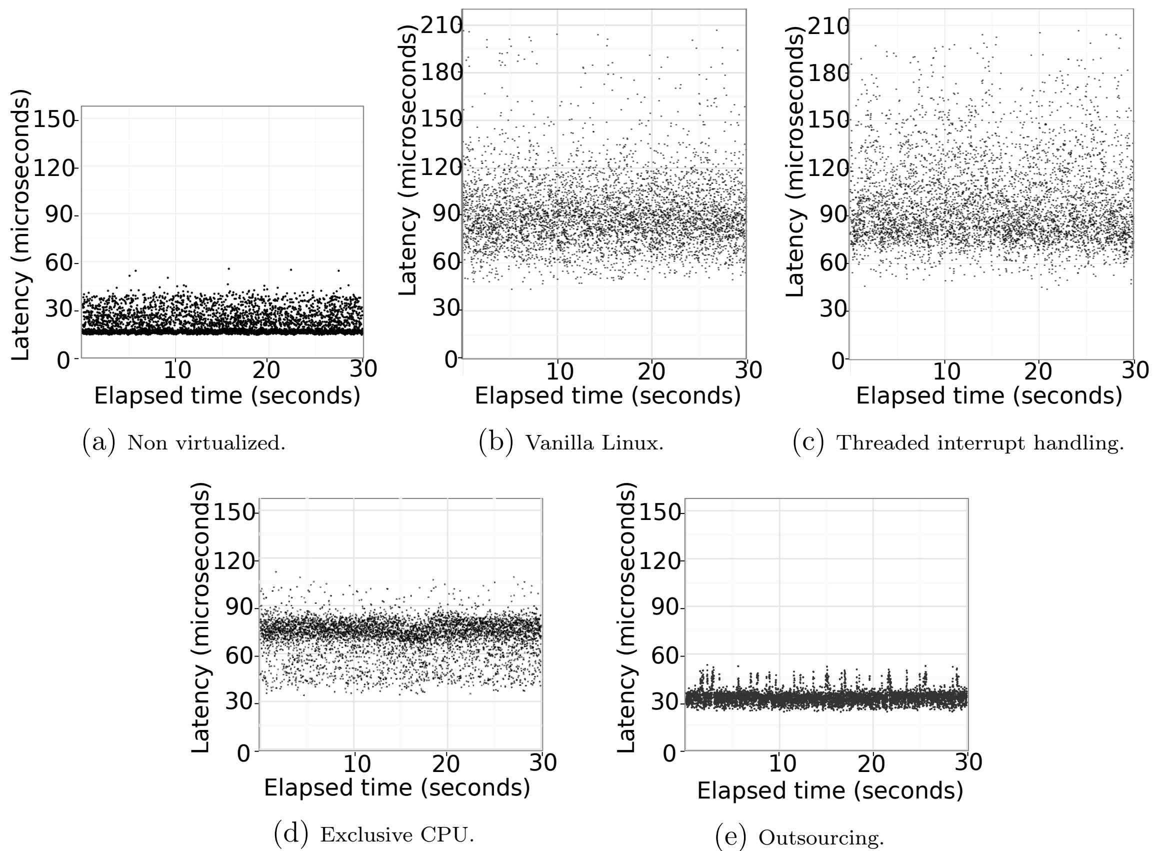
(a) Non virtualized. (b) Vanilla Linux. (c) Threaded interrupt handling.

(d) Exclusive CPU. (e) Outsourcing.

Figure 6-2: Distribution of the Critical RT server response times without running Heavy Receivers.

Table 6.3: Statistical values of the Critical RT server response times without running Heavy Receivers (microseconds).

Method	**Mean**	99th **percentile**	**Standard deviation**
Non virtualized	**12.3**	**13.4**	**0.5**
(non-RT) Vanilla Linux	**27.3**	**34.2**	**1.5**
Threaded interrupt handling	**30.7**	**38.1**	**1.5**
Exclusive CPU	**37.4**	**46.1**	**2.3**
Outsourcing	**21.2**	**25.0**	**1.5**



(a) Non virtualized. (b) Vanilla Linux. (c) Threaded interrupt handling.

(d) Exclusive CPU. (e) Outsourcing.

Figure 6-3: Distribution of the Critical RT server response times with Heavy Receivers.

Table 6.4: Statistical values of the Critical RT server response times with Heavy Receivers (microseconds).

Method	Mean	99th percentile	Standard deviation
Non virtualized	20.5	39.7	6.7
(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

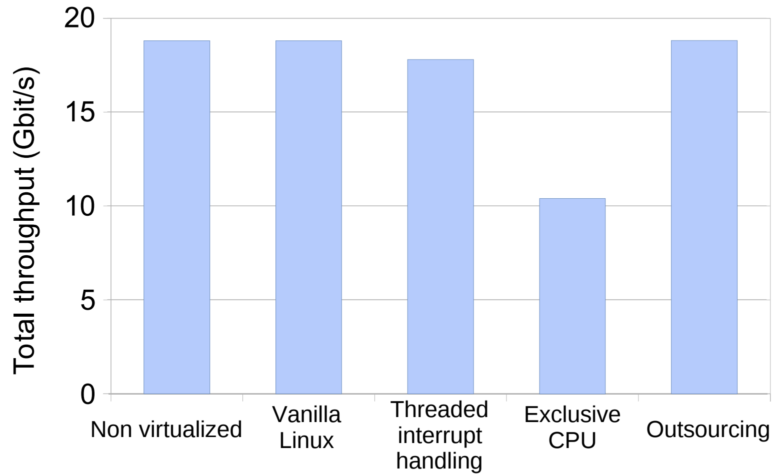


Figure 6-4: Total throughput of Heavy Receivers.

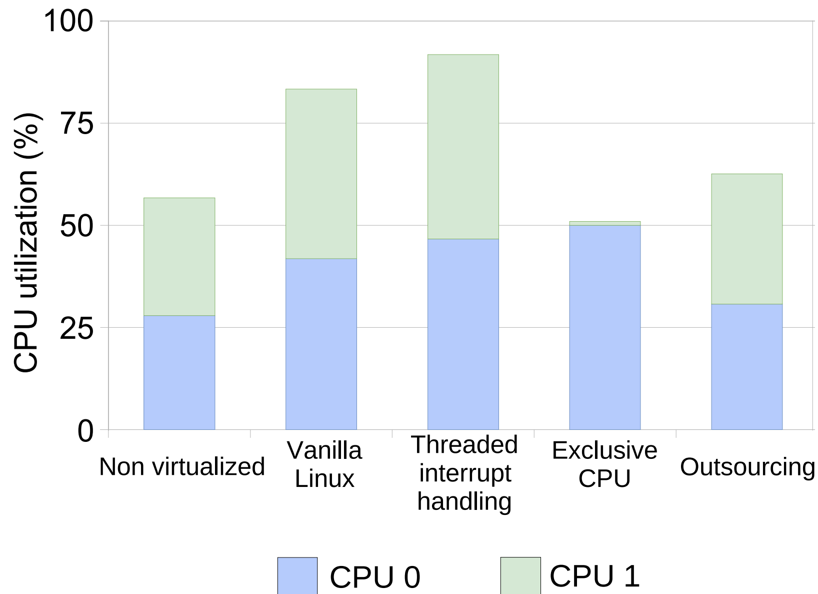
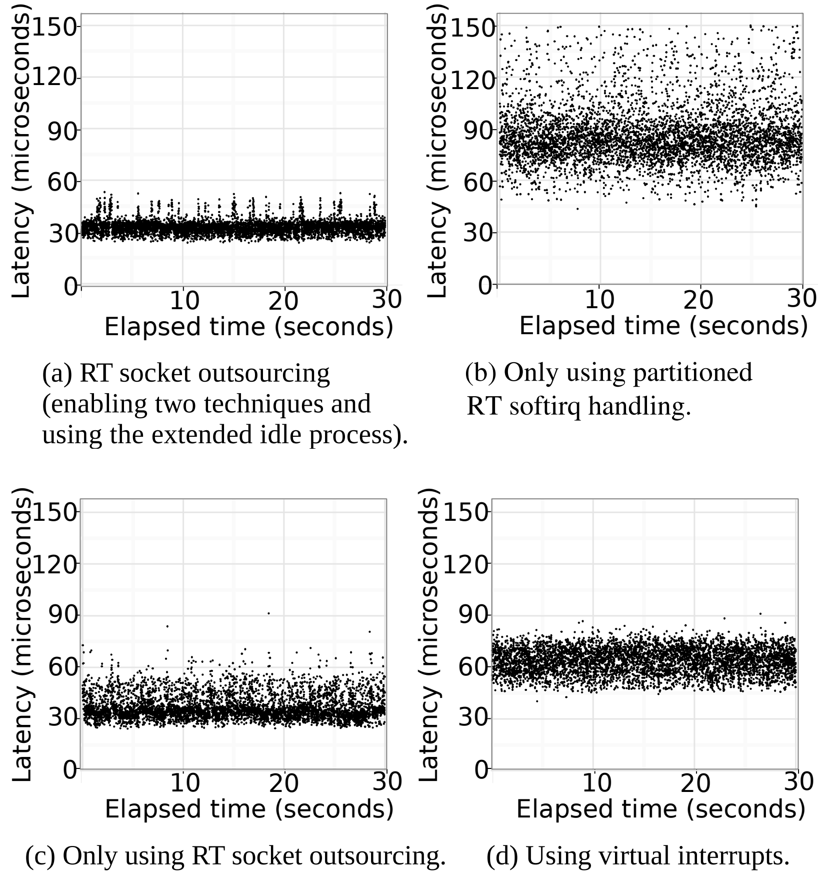


Figure 6-5: Achievable CPU utilization.

6.3 Effects of individual techniques

The outsourcing method comprises two techniques: partitioned RT softirq handling (Chapter 4) and RT socket outsourcing (Chapter 5). We performed the same experiments as in Section 6.1 by enabling one of two techniques at a time. Figure 6-6a shows the result using both techniques, and Figures 6-6b and 6-6c show the results using one of the two techniques without the other. Table 6.5 summarize the statistical values (the mean, 99 th percentile, and standard deviation (SD)). When we enabled



(a) RT socket outsourcing (enabling two techniques and using the extended idle process).

(b) Only using partitioned RT softirq handling.

(c) Only using RT socket outsourcing.

(d) Using virtual interrupts.

Figure 6-6: Distribution of the Critical RT server response times with the outsourcing method using individual techniques.

Table 6.5: Statistical values of the Critical RT server with the outsourcing method using individual techniques.

Method	Mean	99th percentile	Standard deviation
RT socket outsourcing (enabling two techniques and using the extended idle process)	32.9	46.8	4.3
Only using partitioned RT softirq handling.	83.3	155.2	19.3
Only using RT socket outsourcing	34.8	60.7	7.4
Using virtual interrupts	64.1	79.1	9.0

only one of the two techniques, we obtained larger variances of the response times than those obtained using both techniques.

In Section 5.2, we described the extended idle process to eliminate virtual interrupt handling from a guest OS. We compared these two mechanisms using the same

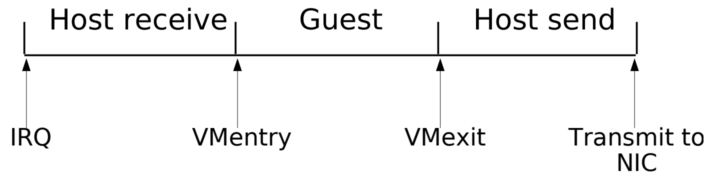


Figure 6-7: Division of the message processing path into three segments.

experiments as in Section 6.1. Figure 6-6a and Figure 6-6d show the results. Using the extended idle process (Figure 6-6a) showed better real-time characteristics than using virtual interrupts (Figure 6-6d). Compared to using virtual interrupts, the extended idle process reduced the standard deviation by a factor of 2 μs (Table 6.5).

6.4 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. We divided the processing path into the following segments (Figure 6-7) for virtualized environments, in the same way as in Section 3.4.2:

- **Host receive:** Host execution from the receipt of an IRQ to the start of the guest OS execution (VM entry). This segment includes the network stack processing and the execution of the network device backend thread (vNIC thread).
- **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.

While the non virtualized environment does not have the idea of the host and the guest, we divided the processing path into the following segments for comparison.

- **Host receive:** Kernel execution from the receipt of an IRQ to the start of the user process execution.

- **Guest:** User process execution to the invocation of the sendto() system call.
- **Host send:** Kernel execution from the invocation of the sendto() system call 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 6.1.

Figure 6-8 presents the experimental results in an environment without running the Heavy Receivers. The figure shows low latency variance in all the methods when there are no co-located non-RT servers.

Figure 6-9 presents the results of these experiments where the critical RT server was co-located with two Heavy Receivers. Figure 6-9a shows that the non virtualized environment had high latency variances in the "host receive" segment. By comparing three columns of Figure 6-9, 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 in the threaded interrupt handling method, as described in Section 3.4. These priority inversion problems were not present when using the exclusive CPU method, and when using the outsourcing method, as described in Chapter 4.

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.

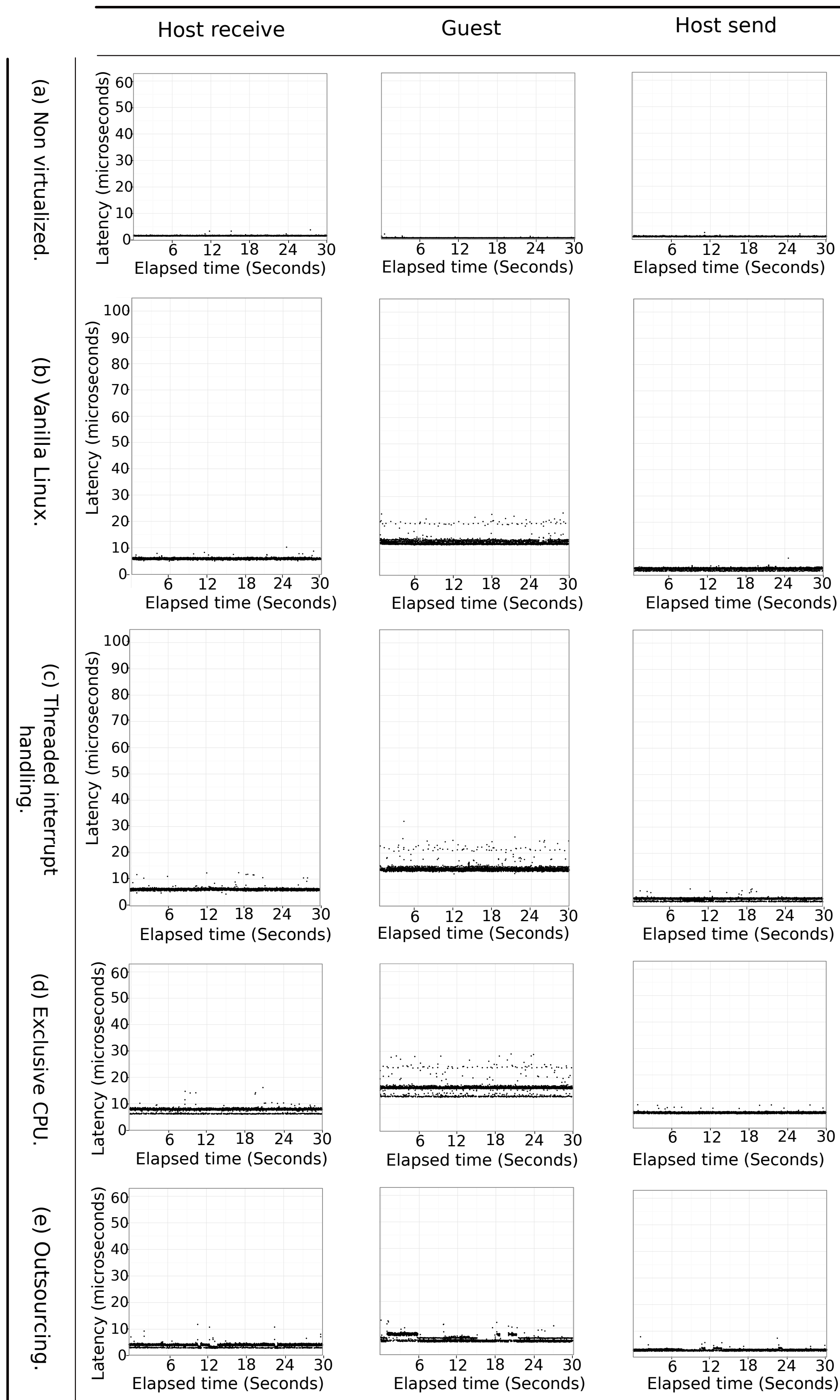


Figure 6-8: Latencies in three segments of the processing path of RT messages without running Heavy Receivers.

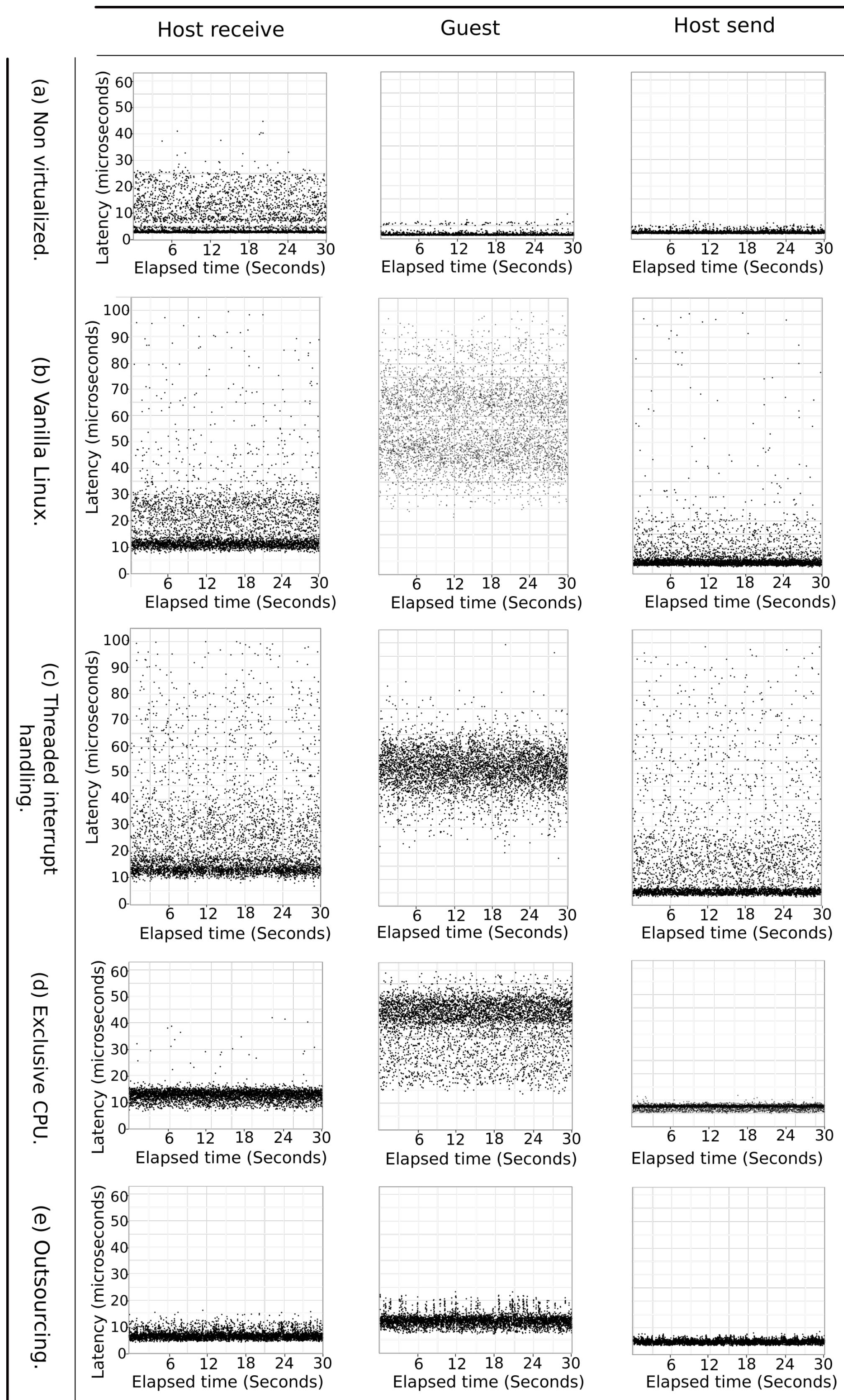


Figure 6-9: Latencies in three segments of the processing path of RT messages with running Heavy Receivers.

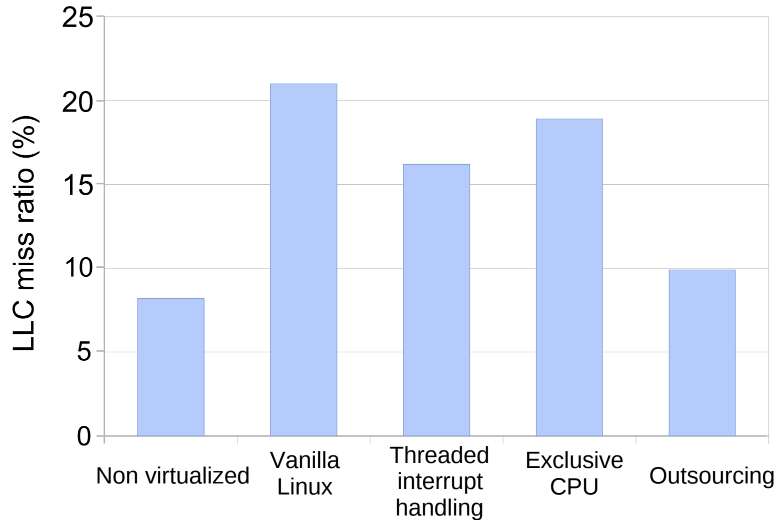
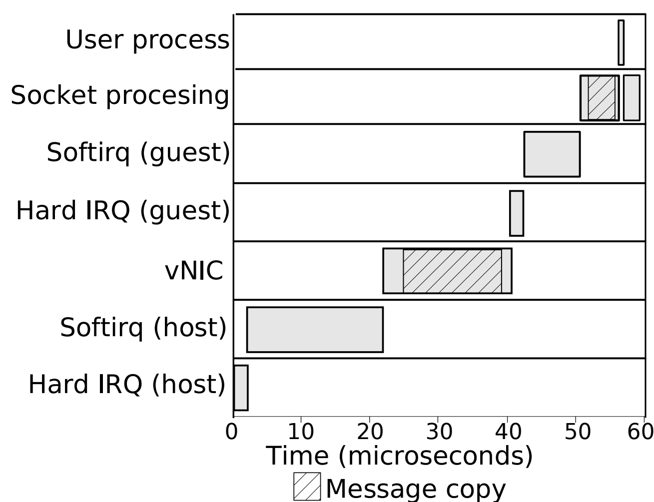


Figure 6-10: LLC miss ratio of the RT threads.

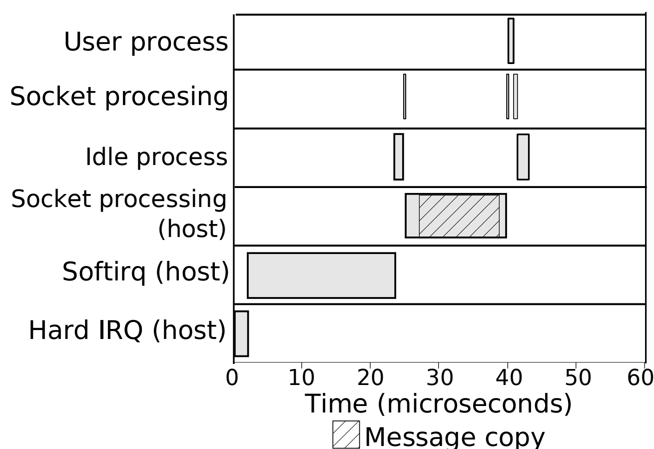
6.5 Cache pollution in the RT methods

Using the same experiments as in Section 6.1, we analyzed the impact on the LLC. We obtained the numbers of cache references and misses using the perf command of Linux [60]. 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 6-10 presents the results of these experiments. The cache pollution when using the outsourcing method was the lowest among vanilla Linux and the two conventional RT methods because message processing had a smaller memory footprint. The outsourcing method and non-virtualized environment had similar cache miss ratios because the number of message copying of the Heavy Receivers were same. 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. Note that compared to vanilla and the conventional RT methods, the outsourcing method had an LLC miss ratio close to that in the non-virtualized environment.



(a) Threaded interrupt handling.



(b) Outsourcing.

Figure 6-11: Message processing path of a non-RT Heavy Receiver.

6.6 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 6.1.

Figure 6-11 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 6.4. In Figure 6-11, “User process”

and "Socket processing" mean the execution of the Heavy Receiver and system call layer in a kernel. Figure 6-11b 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 6-11, message copying is marked with diagonal lines.

Figure 6-11a 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 6-11b 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.

6.7 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 6.1.

6.7.1 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 [48], which exchanges messages based on the Session Initiation Protocol (SIP) [81]. We ran two SIPp [30] 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 10X2-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 of the following six messages by the VoIP server.

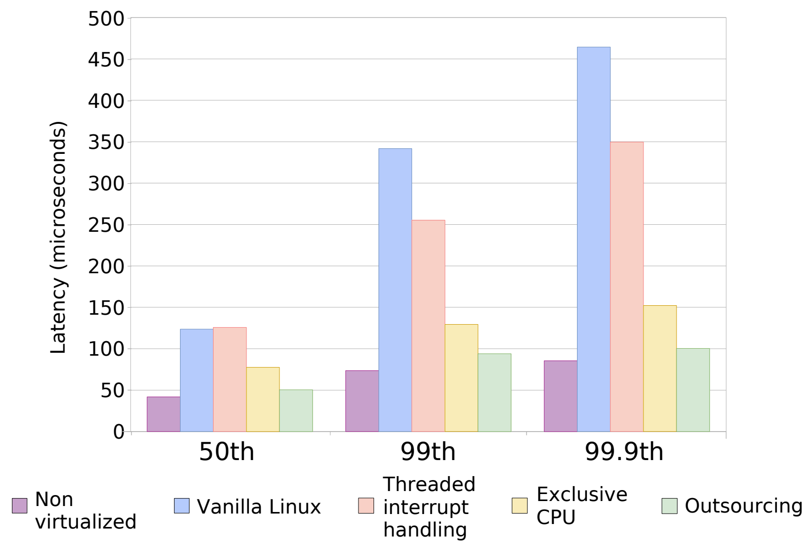
1. An INVITE message from the UAC to UAS. (At this time, the server also sent a TRYING message to the UAC.)
2. A RINGING message from the UAS to the UAC.
3. An OK message from the UAS to the UAC.
4. An ACK message from the UAC to the UAS.
5. A BYE message from the UAC to the UAS.
6. An OK message from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

Similar to using the modified netperf client in Section 6.1, 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. The total incoming and outgoing throughput of SIP were 2.4 Mbps. We ran the same Heavy Receivers employed in the previous experiments.

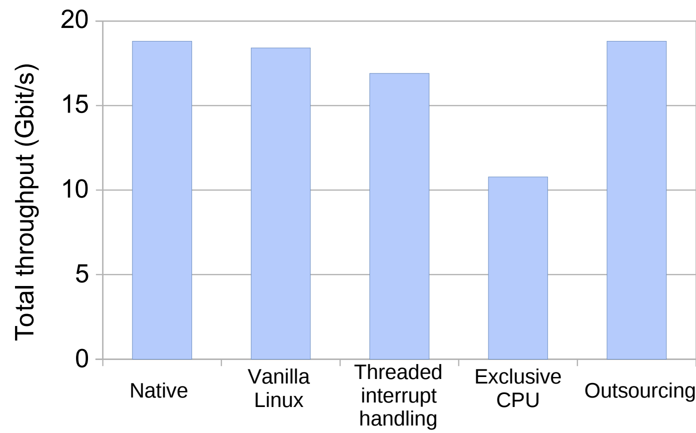
Figure 6-12 shows the experimental results. Figure 6-12a illustrates the percentiles (50th, 99th, and 99.9^{th}) of the forward delays. At the same time, we measured the total throughputs of the Heavy Receivers and the CPU utilization of the VM host. These results are shown in Figure 6-12b and Figure 6-12c, respectively.

Figure 6-12a shows that the outsourcing method had the lowest tail latencies among the RT methods and comparable results with the non-virtualized environment. In the 99th percentiles 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.

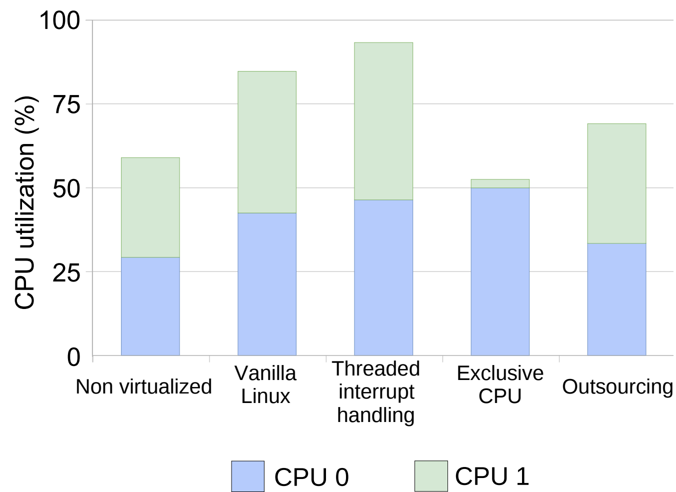
In terms of the total throughputs of the Heavy Receivers and the CPU utilization, the outsourcing method had 13% higher throughput and 15% lower CPU utilization compared to the threaded interrupt handling method. Compared to the exclusive CPU method, the outsourcing method had a higher total throughput by a factor of 2.



(a) Latency of the Critical RT server.



(b) Total throughput of Heavy Receivers.



(c) Achievable CPU utilization.

Figure 6-12: The results of application benchmark using a VoIP server.

6.7.2 Memcached

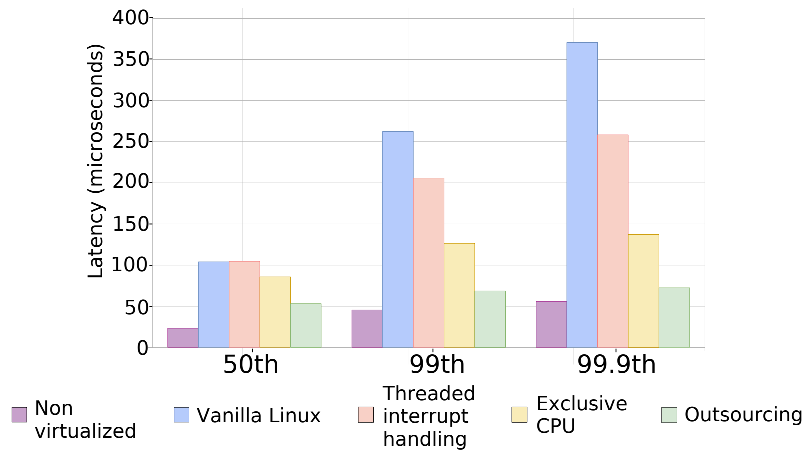
We ran Memcached [21] as a Critical RT server. Memcached is a distributed key-value store that is widely used for caching. The benchmark program was memaslap [1], which was executed in a remote machine.

Similar to using the modified netperf client in Section 6.1, we modified memaslap. The modified 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. The total incoming and outgoing throughput of Memcached were 364 Kbps and 1.8 Mbps, respectively. We measured the response times of the GET requests using the hardware monitor (Endace DAG 10X2-S card). We ran the same Heavy Receivers employed in the previous experiments.

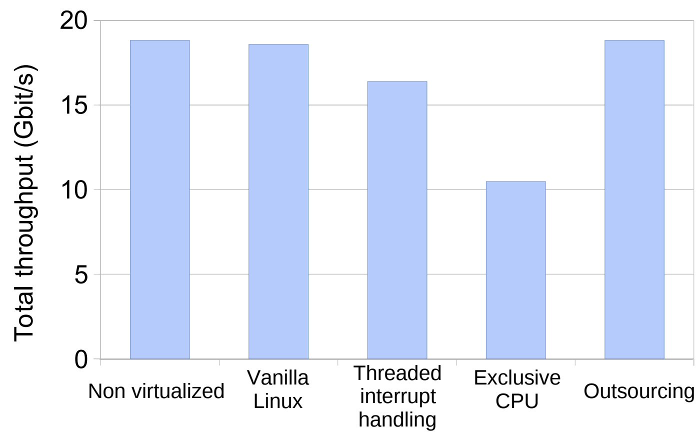
Figure 6-13 presents the experimental results. Figure 6-13a illustrates the percentiles (50th, 99th, and 99.9^{th}) of the response times. At the same time, we measured the total throughputs of the Heavy Receivers and the CPU utilization of the VM host. These results are shown in Figure 6-13b and Figure 6-13c, respectively.

The outsourcing method produced the best results among the RT methods and comparable results with the non-virtualized environment. In Figure 6-13a, the outsourcing method had 74%, 67% and 46% lower latency than vanilla Linux, the threaded interrupt handling method and the exclusive CPU method, respectively.

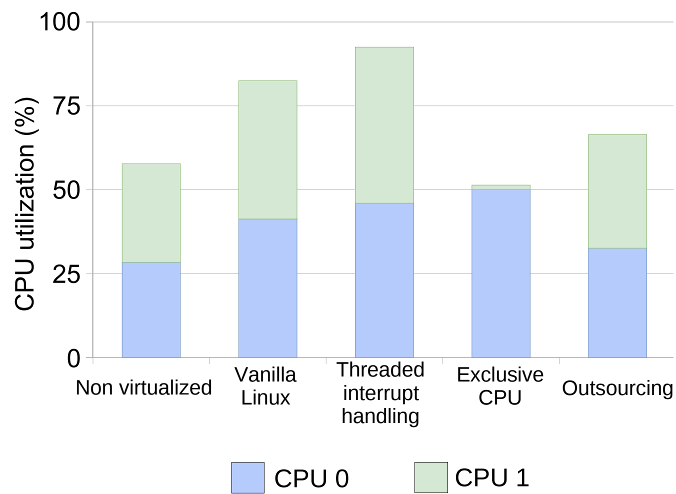
As presented in Figure 6-13b, the outsourcing method achieved the same high throughput of 18.8 Gbps as the non virtualized environment and Vanilla Linux. Compared to the threaded interrupt handling method, the outsourcing method had 13% higher throughput and 26% lower CPU utilization. Compared to the exclusive CPU method, the outsourcing method had a higher total throughput by a factor of 2.



(a) Latency of the Critical RT server.



(b) Total throughput of Heavy Receivers.



(c) Achievable CPU utilization.

Figure 6-13: The results of application benchmark using memcached.

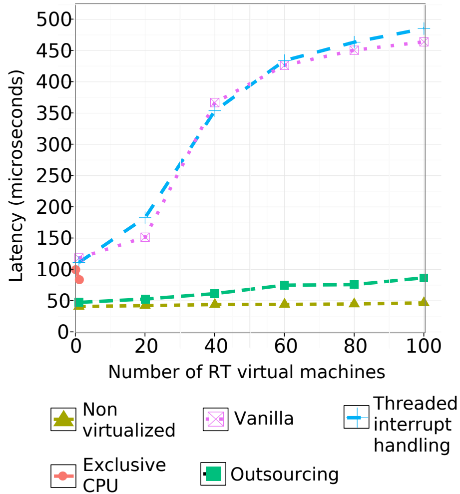
6.8 Scalability of RT virtual machines

In Sections 6.2 to 6.7, 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.

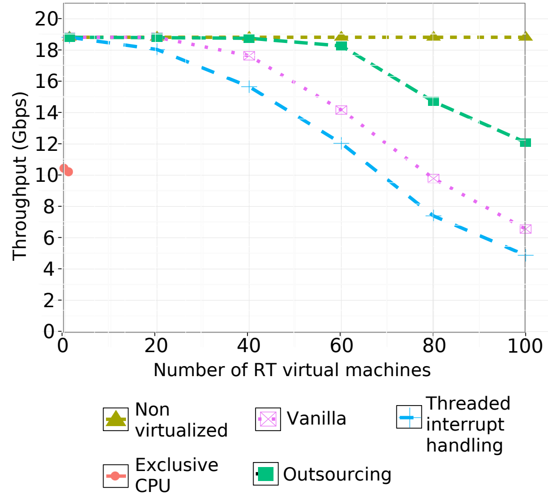
In this section, we performed experiments using the same configuration as in the previous sections except that we set the number of active CPU cores to four. We ran a Heavy Receiver in a single non-RT VM, and we ran two non-RT VMs as in the previous sections. We ran a Critical RT server in a single RT VM and increased the number of RT VMs up to 100 for the threaded interrupt handling method and the outsourcing method, as well as the non virtualized environment and vanilla Linux. We increased the number of RT VMs 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 6.2. 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 10X2-S card). We ran a netperf, VoIP, or memcached server as a Critical RT server in an RT VM. We ran an iperf server as a Heavy Receiver in an non-RT VM. For comparisons, we also run these servers in a non virtualized environment, that is, the host OS.

Figure 6-14 shows the experimental results using a netperf server as a Critical RT server. In these figures, the x-axis represents the number of RT VMs. Figure 6-14a shows the 99th percentiles of the Critical RT response times, Figure 6-14b shows the total throughput of the non-RT Heavy Receivers, and Figure 6-14c shows the achievable CPU utilization.

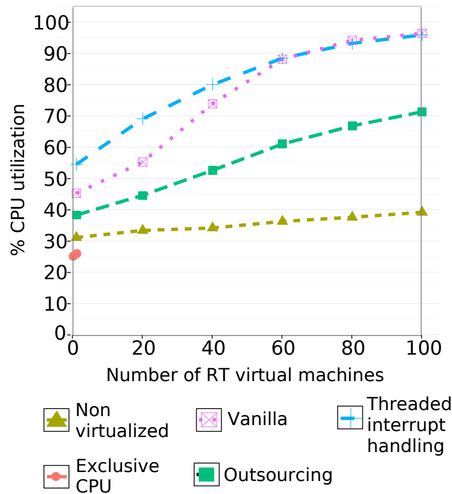
As shown in Figure 6-14a, the non virtualized environment scaled well and the exclusive CPU method did not scale. Both the threaded interrupt handling method and the outsourcing method scaled up to 100 VMs. The outsourcing method produced much smaller latency variances of the Critical RT servers (around two times of the non virtualized environment) than the threaded interrupt handling method (and vanilla



(a) 99th percentile latency of Critical RT servers.



(b) Total throughput of Heavy Receivers.

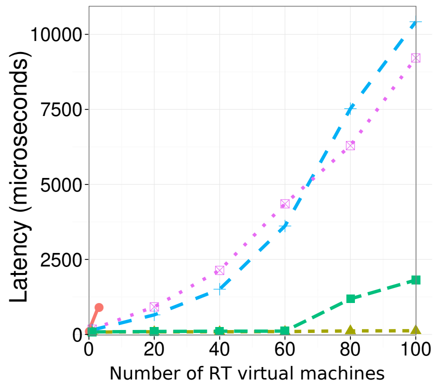


(c) Achievable CPU utilization.

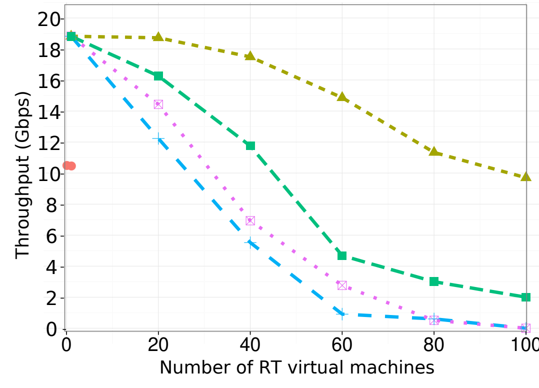
Figure 6-14: Scaling the number of RT virtual machines (Netperf as critical RT server).

Linux). As shown in Figures 6-14b and 6-14c, 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 6-14b.

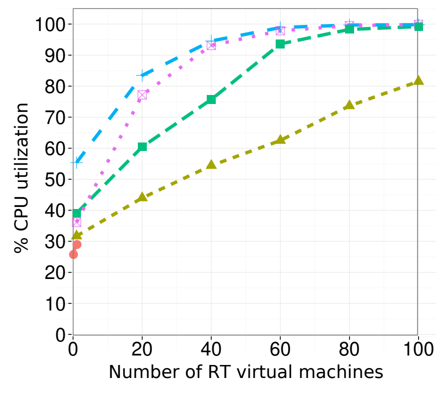
We also performed similar experiments using a VoIP server and memcached server as a Critical RT server. The experimental results are presented in Figure 6-16 and Figure 6-15, respectively. In Figure 6-15a, the outsourcing method scaled well up to the 60 RT VMs. When the number of the RT VMs was 60 VMs, the threaded



(a) 99th percentile latency of Critical RT servers.



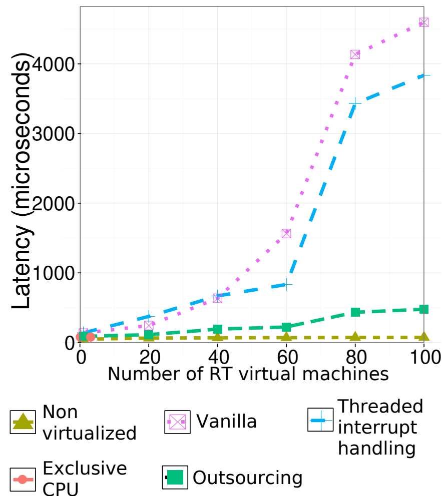
(b) Total throughput of Heavy Receivers.



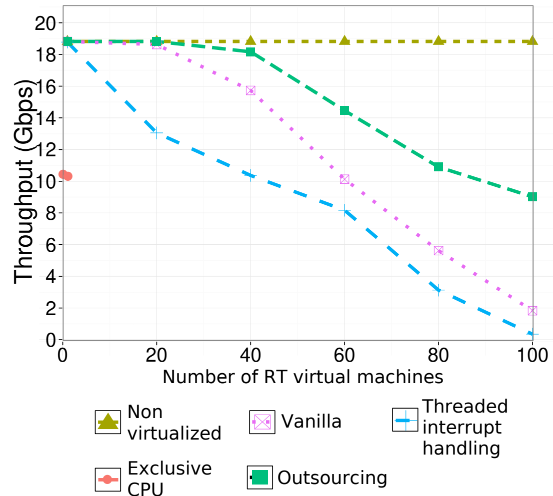
(c) Achievable CPU utilization.

Figure 6-15: Scaling the number of RT virtual machines (VoIP server as critical RT server).

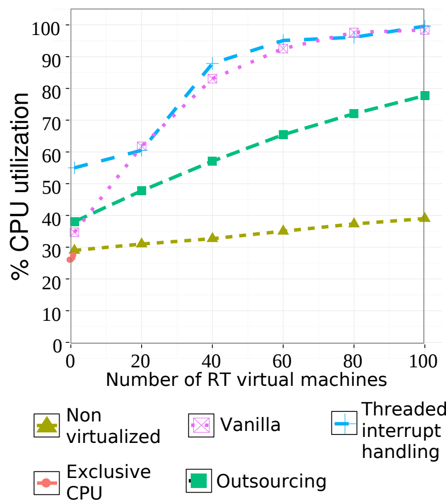
interrupt handling method and the outsourcing method had a CPU utilization of 95% and 94% respectively. At this number of VMs, the 99th percentile latency of the Critical RT server in the outsourcing method was 111.2 μs. This same latency in the threaded interrupt handling method exceeded 3.5 ms. Since the outsourcing method has a lower memory footprint compared to the conventional RT methods, the throughput of the Heavy Receivers decreased more slowly when increasing the number of RT VMs. When the number of RT VMs was 60, the throughput of the Heavy Receivers in the outsourcing method was 4.6 Gbps, while that in the threaded



(a) 99th percentile latency of Critical RT servers.



(b) Total throughput of Heavy Receivers.

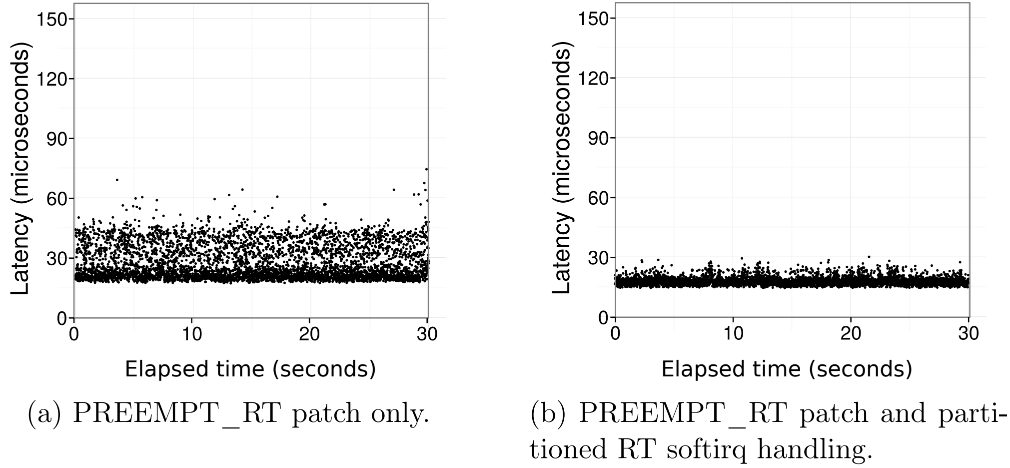


(c) Achievable CPU utilization.

Figure 6-16: Scaling the number of RT virtual machines (Memcached as critical RT server).

interrupt handling method was less than 1 Gbps. When the number of RT VMs was greater than 60, the required CPU resources by the Critical RT servers exceeded the available CPU resources and the latency increased rapidly.

Figure 6-16 shows the experimental results using a memcached server as a Critical RT server. This is similar to Figure 6-15 using a VoIP server as a Critical RT server. Compared to the VoIP server, the memchached server had a lower throughput and lower CPU utilization than the VoIP server, as described in Section 6.7. In terms of 99th percentile latency of the Critical RT server, the outsourcing method scaled up



(a) PREEMPT_RT patch only.

(b) PREEMPT_RT patch and partitioned RT softirq handling.

Figure 6-17: Distribution of the Critical RT server in container-based virtualization.

Table 6.6: Statistical values of the Critical RT server response times with running Heavy Receivers in container-based virtualization (microseconds).

Method	Mean	99th percentile	Standard deviation
PREEMPT_RT patch only	**26.9**	**48.8**	**8.6**
PREEMPT_RT patch and partitioned RT softirq handling	**17.8**	**24.6**	**1.9**

to 100 RT VMs. At the same time, the CPU utilization was around 80% and the throughput of the Heavy Receivers was 9 Gbps.

6.9 Using partitioned RT softirq handling in container-based virtualization

The outsourcing method comprises two techniques: partitioned RT softirq handling (Section 4.1) and RT socket outsourcing (Section 5.2). We can use the former technique in container-based virtualization because an individual container shares the network stack of the host OS. In this subsection, we evaluate this technique in container-based virtualization.

To evaluate the partitioned RT softirq handling technique in container-based virtualization, we repeated the experiment in Section 6.1 and execute the Critical RT server and the Heavy Receivers in a Docker container [23]. As similar to the threaded

interrupt handling method, we use the PREEMPT_RT patch for Linux in the host OS.

Figure 6-17 and Table 6.6 present the experimental results. Figure 6-17a shows the distribution of the latency of the Critical RT server using only the PREEMPT_RT patch. Figure 6-17b shows that using the partitioned RT softirq handling technique along with the PREEMPT_RT patch. As shown in these figures and Table 6.6, the partitioned RT softirq handling technique effusively reduced latency and latency variances. Using only the PREEMPT_RT patch, 99th percentile latency was 48.8 μs and the standard deviation was 8.9 μs. By enabling the partitioned RT softirq handling technique, 99th percentile latency was reduced to 24.6 μs and the standard deviation was reduced to 3.9 μs.

6.10 Current restrictions and limitations

The outsourcing method requires changing the kernel code of the host and guests. This poses some limitations. Partitioned RT softirq handling is dependent on the current implementation of Linux networks stacks. We should send this patch to the developer group of the PREEMPT_RT patch. To use the outsourcing method in a new guest OS, we have to change the system call layers and the idle process. Previous work shows that this changing does not require a large effort. For example, socket outsourcing was also implemented in NetBSD and Windows guests [27, 52].

The outsourcing method comprises two techniques: partitioned RT softirq handling (Section 4.1) and RT socket outsourcing (Section 5.2). RT socket outsourcing method is not needed in container-based virtualization because processes in containers utilize the host OS's network stack. In Section 6.9, we have confirmed that partitioned RT softirq handling reduces the latency variance in container-based virtual environments.

In terms of VM management, it is not trivial to migrate VMs that use the outsourcing method because we need to obtain the states in the host kernel. We can obtain the states in the host kernel and implement VM migration using the Check-

point/Restore In Userspace (CRIU) tool [19].

Chapter 7

Conclusion

In this thesis, we described real-time network stacks of commodity hosted virtual machine environments.

Chapter 1 described the background, and problem, and objectives of this research. Real-time (RT) and time-sensitive systems are becoming pervasive. Data centers, for example, host time-sensitive network servers, such as voice-over IP (VoIP) servers and web search engines. Recently, it is a common practice to run real-time applications on commodity OSs, such as Linux. Especially complex real-time systems are often built on commodity OSs because commodity OSs provide rich networking and graphics APIs. In this thesis, we define *real-time* applications as the applications that work well in non-virtualized or native commodity OSs.

Commodity OSs cannot provide sufficient real-time capabilities for some real-time applications. To address this problem and realize low consistent latencies, developers and researchers are developing extensions or patches to commodity OSs.

Commodity OSs continuously evolve and become complex and this causes an inherent problem. That is, most of these changes favor high throughput over low latency and small variance of latency. Because improving real-time capabilities of a commodity OS often decreases throughput, it is usually developed by a separated group as a patch. Every time the mainline group of developers releases a new version of the commodity OS for improving throughput, the group of RT developers makes an RT patch for the new version. This development race is persistent. In this thesis, we

chose a crucial problem from this development race. That is, we achieve a real-time network stack in commodity virtual machine environments.

When data centers host RT servers, they often host non-RT servers in shared hardware platforms using virtual machines. Such data centers give higher priorities to the RT servers. Nonetheless, non-RT servers can interfere with RT servers and cause variances of response times of the RT servers. It is not trivial to find the causes of these problems because the network stack of the target environment is complex and evolving. Most existing systems do not solve this problem but bypass the problem. Data centers often allocate exclusive physical resources to RT servers with sacrificing CPU utilization.

The research objective of this thesis is to achieve the following goals at the same time:

- Achieve short and consistent latency for RT servers.
- Obtain high throughput for non-RT servers and avoid low CPU utilization within the bound of the consistent latency for RT servers.

To accomplish this objective, 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.

Chapter 2 described related work. Early systems, such as Lazy receiver processing (LRP) in 4.4 BSD [24], and prioritized interrupt handling in Solaris [53] add real-time capabilities to these commodity OSs. RTLinux [5], Time-Sensitive Linux [34], Resource Kernel [33], and Xenomai [32] add real-time capabilities to the Linux kernel. While these systems effectively reduced latency variance, the persistent evolution of the base operating system code introduces new sources of variance. In this thesis, we have tackled new sources of variance. While RT-Xen [94] improves real-time capability of Xen by replacing the scheduler, its absolute real-time performance is poor. Although techniques in [67, 76] increase the throughput by avoiding message copying, they do not consider latency and latency variance.

Socket-outsourcing and similar techniques in [27, 29] offload guests' high-level socket operations to the host. These techniques improve throughput by eliminating message copying and by sending TCP acknowledgment packets efficiently. In this thesis, we adopt socket outsourcing to eliminate message copying and mitigate the cache pollution problem. Some techniques such as Polling threads [62], Netmap [80] and DPDK [59], enable user space applications to send and receive packets by polling mode and eliminate overheads of interrupt handling in operating systems. To use these techniques, network applications using the socket API should be modified. In this thesis, we run unmodified network applications and achieve consistent latency. Finally, techniques such as [35,36,90] use advanced hardware facilities to improve I/O performance in VM environments. For example, paper [2] uses Single-root input/output virtualization (SR-IOV) [75] and reduces the latency and latency variance. This thesis proposes a software-based method for consistent latency.

In Chapter 3, we illustrated the causes of latency variances in vanilla Linux and two conventional RT methods: the threaded interrupt handling method and the exclusive CPU method. We used a typical virtual machine environment that hosted two types of servers: the Critical RT server which is an RT server and receives requests from clients occasionally and sends response messages to the clients, and the Heavy Receiver which is a non-RT server and receives messages persistently from clients at the maximum speed and stresses the receiver-side of the network stack. We measured the latency, that is, the response times of the Critical RT server, with a hardware monitor (an Endace DAG 10X2-S card [35]). The exclusive CPU method had the lowest latency variance compared with vanilla Linux and the threaded interrupt handling method. However, the exclusive CPU method has a drawback. Because the CPU dedicated to the Critical RT server did not help to execute non-RT Heavy receivers, this method yielded less throughput of the Heavy receiver and less achievable CPU utilization.

We divided the message processing path of the Critical RT server into segments and inserted our own lightweight probes. We identified that most of the latency variances were located in the "host receive" segment and the "guest" segment. We confirmed a priority inversion in the interrupt-first host kernel of vanilla Linux. We

have found another new priority inversion in the softirq mechanism of the threaded interrupt handling method. The latency variances in the "guest" segment were caused by the LLC pollution by the non-RT Heavy Receivers. Not only the threaded interrupt handling method but also the exclusive CPU method have this problem.

In Chapter 4, we described one of two techniques of our proposed method. We call this technique *partitioned RT softirq handling*. Similar to the threaded interrupt handling method, this technique avoids the priority inversion problem in the interrupt-first host kernel by using the PREEMPT_RT patch and assigning high priorities to RT threads. Second, this technique avoids the priority inversion problem in the host's softirq handling by dividing softirq handling into RT and non-RT types. We divided the poll_list of the softirq mechanism into two types: the (non-RT) poll_list for non-RT softirq handlers, and rt_poll list for RT softirq handlers. Similarly, we divide the softirq lock into two locks: (non-RT) softirq_lock and *rt_softirq_lock*. In this technique, non-RT softirq handlers run with the same priority of the non-RT interrupt handlers threads and can be preempted by the RT softirq handlers and RT threads.

Chapter 5 described the other technique of our proposed method. We call this technique *RT socket outsourcing*. We extended conventional socket outsourcing to overcome the cache pollution problem. Socket outsourcing allows a guest kernel to delegate high-level network operations to the host kernel. When a guest process invokes a socket operation, its processing is delegated to the host. In socket outsourcing, the incoming network messages going to a guest process are handled by the host network stack. While message copying is performed two times in conventional RT methods, in socket outsourcing, message copying is performed once. This omission of copying makes the footprint smaller and reduces the cache pollution by non-RT servers. This lowers latency variance of RT servers.

Conventional socket outsourcing can face the priority inversion problem in the softirq mechanism in a guest because it makes use of virtual interrupts. We solved this problem by removing interrupt handling from a guest OS for receiving RT messages in RT socket outsourcing. We implement this by extending the idle process of Linux. This idle process in the guest OS executes a halt instruction and this places the

vCPU thread into sleep mode in the host. When the host receives a new message, the hypervisor does not inject a virtual interrupt but instead, it resumes the VM. The modified idle process on the guest OS checks the event queue and the states of the sockets in the shared memory. When the modified idle process notices the arrivals of new messages, it wakes up the receiving processes and goes back to the scheduler. The scheduler executes these processes immediately without interrupt handling.

In Chapter 6, we evaluated the outsourcing method by comparing it with the two conventional RT methods. We performed experiments using a simple RT server in the experimental environment. Compared to the threaded interrupt handling method, the proposed method reduced the standard deviation of the latencies of a simple RT server by a factor of 6, and achieved 5.6% higher throughput and 32% lower CPU utilization. Compared to the exclusive CPU method, the proposed method reduced the standard deviation by a factor of 2 and prevented underutilization of the exclusive CPU. Next, We ran a VoIP server as a Critical RT server and measured the forward delays of the VoIP server. The outsourcing method had the lowest tail latencies among the RT methods. In the 99th percentiles results, the outsourcing method had 63% lower latency than the threaded interrupt handling method and 27% lower latency than the exclusive CPU method. We performed another application experiment using Memcached as a Critical RT server. We obtained similar results to that of VoIP. Finally, we increased the number of RT VMs and evaluated the scalability of these methods. We ran a Critical RT server in a single RT VM and increased the number of RT VMs up to 100. The proposed method was more scalable in terms of the number of RT VMs than the conventional RT methods.

In conclusion, it is challenging to achieve a consistent real-time latency in commodity virtual machine environments because they have longer and more complex network protocol stacks. This thesis analyzed such network stacks, found the causes of the problems, and proposed a method that achieved consistent latency in a Linux KVM-based hosted environment. First, this method solved the priority inversion problem in the interrupt-first host kernel of vanilla Linux using the PREEMPT_RT patch. Second, this method solved another priority inversion problem in the softirq

mechanism of Linux by explicitly separating the RT softirq handling from the non-RT softirq handling. Finally, this method mitigated the cache pollution problem by co-located non-RT servers and avoided the second priority inversion in a guest OS by socket outsourcing.

While the RT socket outsourcing can mitigate cache pollution, advanced hardware support can also solve this problem. In the future, we would like to use advanced hardware support, such as Intel's Cache Allocation Technology (CAT) [40] to further avoid the cache pollution problem.

Commodity OSs are evolving and becoming complex. We do not see the end of the development races between improving throughput and achieving consistent latency. Until the end of the development races, it is important for real-time patch developers to find the causes of latency variances. While Linux has many performance measurement tools, they are usually not sufficient for real-time patch developers. They often have non-negligible probe effects and produce too many event logs. In this research, we have inserted our own lightweight probes to a small number of prospective points by hand. We had to repeat experiments as changing probe points. In the future, we should find a systematic method to develop real-time patches with better tools.

Acknowledgments

The ideas in this thesis, its character and its form have been heavily influenced by the help and support of several individuals. First, I am grateful for the patience and for the guidance received from my advisor, Professor Yasushi Shinjo and from Professor Calton Pu. It was a very enriching experience to work with them.

I would like to thank my committee members, Professor Koichi Wada, Professor Kazuhiko Kato, Professor Akihisa Ohya, Professor Shuichi Oikawa, and Professor Yoshihiro Oyama. Their comments and suggestions contributed to improve this work.

I also would like to thank developers and researchers that devote their lives to the progress of real-time systems.

During my Ph.D. process, I was supported by the Monbukagakusho Scholarship and by the Monbukagakusho Honors Scholarship, provided by the Ministry of Education, Culture, Sports, Science, and Technology of Japan.

Lastly, I would like to express my gratitude to my family and friends, who supported me from miles away.

List of publications

Publications

- 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 *SCF International Conference on Cloud Computing (CLOUD 2018)*, pages 189-205, 2018.

- O. Garcia, Y. Shinjo, and C. Pu, "Achieving consistent real-time latency at scale in a commodity virtual machine environment through socket outsourcing-based network stacks", *IEEE Access*, Vol. 6, pages 69961-69977, 2018.

Other publications

- Chung-Fan Yang, Oscar Garcia and Yasushi Shinjo, "Achieving consistent real-time latency in a collocated commodity virtual machine environment by LLC partitioning", Workshop on *Efficient Real-time Data Network (ERDN)*, 6 pages, 2018.

Bibliography

[1] B. Aker. "Libmemcached-Memaslap". [Online]. Available: http://libmemcached.org/libMemcached.html, Accessed on: Jun. 3, 2018.

[2] J. Anderson, H. Hu, U. Agarwal, C. Lowery, H. Li, and A. Apon. "Performance considerations of network functions virtualization using containers". In *IEEE Computing, Networking and Communications (ICNC), 2016 International Conference on*, pages 1–7, 2016.

[3] ARM Limited. "ARM 9 Technical Reference Manual - Cache lockdown". [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0201d/I33878.html, Accessed on: Aug. 21, 2018.

[4] M. Aron and P. Druschel. "Soft timers: efficient microsecond software timer support for network processing". *ACM Transactions on Computer Systems (TOCS)*, 18(3):197–228, 2000.

[5] M. Barabanov and V. Yodaiken. "Introducing real-time Linux". *Linux journal*, 34:9, 1997. [Online]. Available: https://www.linuxjournal.com/article/232, Accessed on: Jan 3, 2019.

[6] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio. "Performance comparison of VxWorks, Linux, RTAI, and Xenomai in a hard real-time application". *IEEE Transactions on Nuclear Science*, 55(1):435–439, 2008.

[7] T. Barbette, C. Soldani, and L. Mathy. "Fast userspace packet processing". In *Proceedings of the ACM/IEEE Symposium on architectures for networking and communications systems (ANCS)*, pages 5–16, 2015.

[8] R. Barry. "FreeRTOS reference manual: API functions and configuration options", 2009. [Online]. Available: https://www.freertos.org/Documentation/FreeRTOS_Reference_Manual_V9.0.0.pdf, Accessed on: Dec. 10, 2018.

[9] S. Baruah, V. Bonifaci, G. D'angelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie. "Preemptive uniprocessor scheduling of mixed-criticality sporadic task systems". *Journal of the ACM (JACM)*, 62(2):14, 2015.

[10] D. Beal, E. Bianchi, L. Dozio, S. Hughes, P. Mantegazza, and S. Papacharalambous. Rtai: Real-time application interface. *Linux Journal*, 29(10), 2000. [Online]. Available: `https://www.linuxjournal.com/article/3838`, Accessed on: Jan. 3, 2019.

[11] A. Burns and R. Davis. "Mixed criticality systems: A review". *Tech. Rep, Department of Computer Science, University of York*, pages 1–69, 2013.

[12] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. "BBR: Congestion-based congestion control". *ACM Queue*, 14(5):50, 2016.

[13] Y. Cheng, Z. Chai, and A. Anwar. "Characterizing co-located datacenter workloads: An Alibaba case study". *arXiv*, 2018.

[14] L. Cherkasova, D. Gupta, and A. Vahdat. "Comparison of the three CPU schedulers in Xen". *SIGMETRICS Performance Evaluation Review*, 35(2):42–51, 2007.

[15] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica. "HUG: multi-resource fairness for correlated and elastic demands". In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 407–424, 2016.

[16] M. Christofferson. "4 ways to improve performance in embedded Linux systems". In *Korea Linux Forum*, 2013.

[17] D. Comer. "Computer networks and internets", Pearson, 2015.

[18] A. Crespo, I. Ripoll, and M. Masmano. "Partitioned embedded architecture based on hypervisor: The xtratum approach". In *IEEE European Dependable Computing Conference (EDCC)*, pages 67–72, 2010.

[19] CRIU project. "Checkpoint/Restore In Userspace (CRIU)", 2018. [Online]. Available: `https://www.criu.org/Main_Page`, Accessed on: Oct. 18, 2018.

[20] N. L. da Fonseca and R. Boutaba. "Cloud services, networking, and management", 2015.

[21] Danga Interactive. "Memcached - A distributed memory object caching system", 2015. [Online]. Available: `https://memcached.org/`, Accessed on: Jun. 3, 2018.

[22] C. Delimitrou and C. Kozyrakis. "Quasar: Resource-efficient and QoS-aware cluster management". *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[23] Docker Inc. "Docker: Enterprise container platform". [Online]. Available: `https://www.docker.com/,Accessedon:Dec.17,2018.`

[24] P. Druschel and G. Banga. “Lazy Receiver Processing (LRP): A network subsystem architecture for server systems”. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 261–275, 1996.

[25] E. Dumazet. “Busy polling: past, present, future”. In *The Technical Conference on Linux Networking*, pages 1–4, 2017.

[26] F. C. Eigler. “Problem solving with SystemTap”. In *Proceedings of the Ottawa Linux Symposium*, pages 261–268, 2006.

[27] H. Eiraku, Y. Shinjo, C. Pu, Y. Koh, and K. Kato. “Fast networking with socket-outsourcing in hosted virtual machine environments”. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 310–317, 2009.

[28] Endace Technology Limited. Endace DAG10X2-S datasheet, 2016. [Online]. Available: `https://www.endace.com/dag-10x2-s-datasheet.pdf`, Accessed on: Jun. 3, 2018.

[29] S. Gamage, R. R. Kompella, D. Xu, and A. Kangarlou. “Protocol responsibility offloading to improve TCP throughput in virtualized environments”. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–34, 2013.

[30] R. Gayraud and O. Jacques. “SIPp benchmark tool”, 2014. [Online]. Available: `http://sipp.sourceforge.net/`, Accessed on: Jun. 3, 2018.

[31] B. Gerofi, M. Takagi, A. Hori, G. Nakamura, T. Shirasawa, and Y. Ishikawa. “On the scalability, performance isolation and device driver transparency of the IHK/McKernel hybrid lightweight kernel”. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pages 1041–1050, 2016.

[32] P. Gerum. “Xenomai - Implementing a RTOS emulation framework on GNU/Linux”, 2004. [Online]. Available: `http://www.xenomai.org/documentation/xenomai-2.5/pdf/xenomai.pdf`, Accessed on: Jun. 5, 2018.

[33] S. Ghosh and R. R. Rajkumar. “Resource management of the OS network subsystem”. In *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, pages 271–279, 2002.

[34] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole. “Supporting time-sensitive applications on a commodity OS”. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and implementation*, pages 165–180, 2002.

[35] A. Gordon, N. Amit, N. Har’El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir. “ELI: Bare-metal performance for I/O virtualization”. In *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 411–422, 2012.

[36] A. Gordon, N. Har'El, A. Landau, M. Ben-Yehuda, and A. Traeger. "Towards exitless and efficient paravirtual I/O". In *Proceedings of the 5th ACM Annual International Systems and Storage Conference (SYSTOR)*, pages 1–6, 2012.

[37] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. "Queues don't matter when you can jump them!". In *Proceedings of 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 1–14, 2015.

[38] Z. Huang, R. Ma, J. Li, Z. Chang, and H. Guan. "Adaptive and scalable optimizations for high performance SR-IOV". In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 459–467, 2012.

[39] IEEE. "IEEE standard for information technology- portable operating system interface (POSIX)- Part 1: system application program interface (API)- Amendment J: Advanced real-time extensions [C Language]". *IEEE Std 1003.1j-2000*, 2000.

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

[41] Intel Corporation. "Intel Ethernet Converged Network Adapter X520 Product Brief", 2012. [Online]. Available: `https://www.intel.com/content/www/us/en/ethernet-products/converged-network-adapters/ethernet-x520-server-adapters-brief.html`, Accessed on: Jul. 23, 2018.

[42] Intel Corporation. "Intel 64 and IA-32 architectures software developer's manual". *Volume 3A: System Programming Guide, Part 1*, 1:468, 2016.

[43] Intel Corporation and Linux Foundation. "KVM Enhancements for OPNFV". [Online]. Available on: `http://events17.linuxfoundation.org/sites/events/files/slides/KVM_Enhancements_final%2B.pdf`, Accessed on: Sep. 20, 2018.

[44] "Interactive Realtime Multimedia Applications on Service Oriented Infrastructures (IRMOS)". *The IRMOS Solution*, 2011. [Online]. Available: `https://irmosproject.seagate.com/Default.html`, Accessed on: Oct 27, 2017.

[45] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. "Silo: predictable message latency in the Cloud". In *ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*, pages 435–448, 2015.

[46] Y. Jiang and Y. Liu. "Stochastic network calculus", 2008.

[47] R. Jones. "Netperf", 1996. [Online]. Available: `https://hewlettpackard.github.io/netperf/`, Accessed on: Jun. 3, 2018.

[48] "Kamailio SIP Server Project". Kamailio SIP server. [Online]. Available: url-https://www.kamailio.org/w/, Accessed on: 3 Jun, 2018.

[49] S. J. Kang, J. H. Park, and S. H. Park. "ROOM-BRIDGE: Vertically configurable network architecture and real-time middleware for interoperability between ubiquitous consumer devices in the home". In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 232–251, 2001.

[50] 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 *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 11–20, 2008.

[51] J. Kiszka. "Hard Partitioning for Linux: The Jailhouse Hypervisor", August 2015. [Online]. Available: `https://events.static.linuxfound.org/sites/events/files/slides/LinuxConNA-2015-Jailhouse_0.pdf`, Accessed on: Dec. 10, 2018.

[52] Y. Koh, C. Pu, Y. Shinjo, H. Eiraku, G. Saito, and D. Nobori. "Improving virtualized windows network performance by delegating network processing". In *Proceedings of the 8th IEEE International Symposium on Network Computing and Applications*, pages 203–210, 2009.

[53] F. Kuhns, D. C. Schmidt, and D. L. Levine. "The design and performance of a real-time I/O subsystem". In *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium*, pages 154–163, 1999.

[54] D. Kyriazis, A. Menychtas, G. Kousiouris, M. Boniface, T. Cucinotta, K. Oberle, T. Voith, E. Oliveros, and S. Berger. "A real-time service oriented infrastructure". *GSTF Journal on Computing (JoC)*, 1(2):196–204, 2018.

[55] Y. C. Lee and A. Y. Zomaya. "Energy efficient utilization of resources in cloud computing systems". *The Journal of Supercomputing*, 60(2):268–280, 2012.

[56] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble. "Tales of the tail: Hardware, OS, and application-level sources of tail latency". In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14, 2014.

[57] Z. Lin, Qian and Qi, J. Wu, Y. Dong, and H. Guan. "Optimizing virtual machines using hybrid virtualization". *Elsevier Journal of Systems and Software*, 85(11):2593–2603, 2012.

[58] Linux foundation. "NAPI". [Online]. Available: `https://wiki.linuxfoundation.org/networking/napi`, Accessed on: Jul. 23, 2018.

[59] Linux foundation. "DPDK - Vhost Library", 2018. [Online]. Available: `https://doc.dpdk.org/guides/prog_guide/vhost_lib.html`, Accessed on: Jan. 3, 2019.

[60] Linux foundation. "Perf: Linux profiling with performance counters", 2018. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page, Accessed on: Jun. 10, 2018.

[61] H. Liu. "A measurement study of server utilization in public clouds". In *IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, pages 435–442, 2011.

[62] J. Liu and B. Abali. "Virtualization polling engine (VPE): Using dedicated CPU cores to accelerate I/O virtualization". In *The 23rd ACM International Conference on Supercomputing*, pages 225–234, 2009.

[63] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. "Improving resource efficiency at scale with Heracles". *ACM Transactions on Computer Systems (TOCS)*, 34:6:1–6:33, 2016.

[64] R. Love. "The Linux kernel preemption project", 2000.

[65] Y. Luo. "Network I/O virtualization for cloud computing". *IT professional*, 12(5):36–41, 2010.

[66] Mentor, a Siemens business. "Nucleus AMP". [Online]. Available: https://www.mentor.com/embedded-software/nucleus/amp, Accessed on: Dec 13, 2018.

[67] H. R. Mohebbi, O. Kashefi, and M. Sharifi. "Zivm: A zero-copy inter-VM communication mechanism for cloud computing". *Computer and Information Science*, 4(6):18–27, 2011.

[68] H. Monden. "Introduction to ITRON the industry-oriented operating system". *IEEE Micro*, 7(2):45–52, 1987.

[69] T. Nakajima, T. Kitayama, H. Arakawa, and H. Tokuda. "Integrated management of priority inversion in real-time mach". In *Proceedings of the Real-Time Systems Symposium*, pages 120–130, 1993.

[70] E. Naone. "technology overview conjuring clouds". 112:54–56, 07 2009.

[71] F. Ning, C. Weng, and Y. Luo. "Virtualization I/O optimization based on shared memory". In *IEEE International Conference on Big Data*, pages 70–77, 2013.

[72] A. Nordal, Å. Kvalnes, and D. Johansen. "Paravirtualizing TCP". In *Proceedings of the 6th ACM International Workshop on Virtualization Technologies in Distributed Computing Date (VTDC)*, pages 3–10, 2012.

[73] A. Ø. Nordal, Å. Kvalnes, R. Pettersen, and D. Johansen. "Streaming as a hypervisor service". In *Proceedings of the 7th ACM International Workshop on Virtualization Technologies in Distributed Computing (VTDC)*, pages 33–40, 2013.

[74] Open platform for NFV. "Opnfv-kvmfornfv Documentation". [Online]. Available: https://media.readthedocs.org/pdf/opnfv-kvmfornfv/latest/opnfv-kvmfornfv.pdf, Accessed on: Dic. 12, 2018.

[75] Peripheral component interconnect special interest group (PCI-SIG). "SR-IOV: Single-root input/output virtualization specifications". [Online]. Available: https://pcisig.com/specifications/iov/.

[76] C. Pinto, B. Reynal, N. Nikolaev, and D. Raho. "A zero-copy shared memory framework for host-guest data sharing in KVM". In *IEEE Scalable Computing and Communications (ScalCom)*, pages 603–610, 2016.

[77] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. "Resource kernels: A resource-centric approach to real-time and multimedia systems". In *Multimedia Computing and Networking 1998*, pages 150–165. International Society for Optics and Photonics, 1997.

[78] Red Hat Inc. "Enterprise Linux for Real Time". [Online]. Available: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux_for_Real_Time/7/html/Installation_Guide, Accessed on: Jun. 5, 2018.

[79] Y. Ren, L. Liu, X. Liu, J. Kong, H. Dai, Q. Wu, and Y. Li. "A fast and transparent communication protocol for co-resident virtual machines". In *8th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 70–79, 2012.

[80] L. Rizzo. "Netmap: A novel framework for fast packet I/O". In *Proceedings of the USENIX Conference on Annual Technical Conference*, pages 101–112, 2012.

[81] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. "SIP: Session Initiation Protocol". RFC 3261, 2002. [Online]. Available: https://tools.ietf.org/html/rfc3261.

[82] S. Rostedt. "Ftrace - Function Tracer". [Online]. Available: https://www.kernel.org/doc/Documentation/trace/ftrace.txt, Accessed on: Sep. 25, 2018.

[83] S. Rostedt. "KernelShark - A front end reader of trace-cmd". [Online]. Available: http://rostedt.homelinux.com/kernelshark/, Accessed on: Jun. 10, 2018.

[84] S. Rostedt and D. V. Hart. "Internals of the RT Patch". In *Proceedings of the Linux symposium*, pages 161–172, 2007.

[85] O. Sefraoui, M. Aissaoui, and M. Eleuldj. "OpenStack: toward an open-source solution for cloud computing". *International Journal of Computer Applications*, 55(3):38–42, 2012.

[86] R. Shea and J. Liu. "Network interface virtualization: challenges and solutions". *IEEE Network*, 26(5), 2012.

[87] Silicon Labs. "Micrium µC/OS", 2017. [Online]. Available: `https://www.micrium.com/`, Accessed on: Jan. 4, 2019.

[88] V. Sinitsyn. "Understanding the Jailhouse hypervisor, part 1", 2014. [Online]. Available: `https://lwn.net/Articles/578295/`, Accessed on: Oct. 1, 2018.

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

[90] C.-C. Tu, M. Ferdman, C. Lee, and T. Chiueh. "A comprehensive implementation and evaluation of direct interrupt delivery". In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 1–15, 2015.

[91] R. van Riel. "Real-time KVM from the ground up". In *KVM Forum*, 2016. [Online]. Available: `https://wiki.linuxfoundation.org/_media/realtime/events/rt-summit2016/kvm_rik-van-riel.pdf`, Accessed on: Jan. 3, 2019.

[92] C. Werner, C. Buschmann, T. Jäcker, and S. Fischer. "Bandwidth and latency considerations for efficient SOAP messaging". *International Journal of Web Services Research*, 3(1):49–67, 2006.

[93] Wind River Systems, Inc. "VxWorks", 2019. [Online]. Available: `https://www.windriver.com/products/vxworks/`, Accessed on: Jan. 4, 2019.

[94] S. Xi, M. Xu, C. Lu, L. T. Phan, C. Gill, O. Sokolsky, and I. Lee. "Real-time multi-core virtual machine scheduling in Xen". In *IEEE International Conference on Embedded Software (EMSOFT)*, pages 1–10, 2014.

[95] Xilinx Inc. "Open Asymmetric Multi Processing (OpenAMP)". [Online]. Available: `https://github.com/OpenAMP/open-amp`, Accessed on: Dec 13, 2018.

[96] 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 *ACM International Conference on Management of Data (SIGMOD)*, pages 955–966, 2014.

[97] M. Xu, L. Thi, X. Phan, H.-Y. Choi, and I. Lee. "vCAT: Dynamic cache management using CAT virtualization". In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 211–222, 2017.

[98] E. Zahavi, A. Shpiner, O. Rottenstreich, A. Kolodny, and I. Keslassy. "Links as a Service (LaaS): Guaranteed tenant isolation in the shared cloud". In *Proceedings of the ACM Symposium on Architectures for Networking and Communications Systems*, pages 87–98, 2016.

[99] T. Zhu, D. S. Berger, and M. Harchol-Balter. "SNC-Meister: Admitting more tenants with tail latency SLOs". In *Proceedings of the 7th ACM Symposium on Cloud Computing*, pages 374–387, 2016.

[100] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. "Prioritymeister: Tail latency QoS for shared networked storage". In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14, 2014.

[101] K. M. Zuberi and K. G. Shin. "Design and implementation of efficient message scheduling for controller area network". *IEEE transactions on computers*, 49(2):182–188, 2000.