Implementation and Comparative Evaluation of an Outsourcing Approach to Real-Time Network Services in Commodity Hosted Environments

Oscar Garcia[1(✉)], Yasushi Shinjo[1(✉)], and Calton Pu[2(✉)]

[1] Department of Computer Science, University of Tsukuba, Tsukuba, Ibaraki, Japan
oscar@softlab.cs.tsukuba.ac.jp, yas@cs.tsukuba.ac.jp
[2] College of Computing, Georgia Institute of Technology, Atlanta, GA, USA
calton.pu@cc.gatech.edu

Abstract. Commodity operating systems (OS) often sacrifice real-time (RT) performance (e.g., consistent low latency) in favor of optimized average latency and throughput. This can cause latency variance problems when an OS hosts virtual machines that run network services. This paper proposes a software-based RT method in Linux KVM-based hosted environments. First, this method solves the priority inversion problem in interrupt handling of vanilla Linux using the RT Preempt patch. Second, this method solves 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 mitigates the cache pollution problem by co-located non-RT services and avoids the second priority inversion in a guest OS by socket outsourcing. Compared to the RT Preempt Patch Only method, the proposed method has the 76% lower standard deviation, 15% higher throughput, and 33% lower CPU overhead. Compared to the dedicated processor method, the proposed method has the 63% lower standard deviation, higher total throughput by a factor of 2, and avoids under-utilization of the dedicated processor.

1 Introduction

Large-scale applications with real-time (RT) or stringent quality of service (QoS) requirements, ranging from large distributed systems such as electronic trading, NextGen air traffic control [1], and web-facing e-commerce n-tier applications to small sensors and edge servers in Internet of Things (IoT), smart applications need fast and consistent network services [2–6]. Owing to the variety of environments in which these applications operate, there is a growing need for commodity operating system (OS) with systematic improvements that can satisfy real-time and stringent QoS performance requirements in both speed and consistency. An indication of this trend is the number of real-time operating systems [7] that build on commodity operating systems (Sect. 5).

M. Luo and L.-J. Zhang (Eds.): CLOUD 2018, LNCS 10967, pp. 189–205, 2018.
https://doi.org/10.1007/978-3-319-94295-7_13

Because of the need and effort to maximize average performance in commodity operating systems, they tend to have larger latency variabilities and wider spread in the latency of kernel services, particularly for I/O devices. These latency variabilities are not bugs in the traditional sense, since the system functionality is correctly implemented. These latency variance problems can also cause problems in hosted virtual machines (VM) that run network services.

In this study, we have found three major sources of variance in "vanilla" Linux: two priority inversion problems (Sects. 2.1 and 2.3) and cache pollution by co-located non-RT services (Sect. 2.5). We describe a new approach to real-time network services in Linux KVM-based commodity hosted environments, and evaluate our approach by comparing two current production RT methods. We call our approach the "outsourcing plus separation of RT Softirq" method or the outsourcing method for short. First, we solve the first priority inversion problem in interrupt handling of vanilla Linux using the RT Preempt patch [8]. However, using this patch only has the second priority inversion problem in the softirq mechanism of Linux in the host OS (Sect. 2.3). We can avoid this problem by dedicating a processor exclusively for RT threads (Sect. 2.4). However, this method has disadvantages of low CPU utilization and low total throughput. Therefore, we solve the second priority inversion problem in the host OS by explicitly separating the RT softirq handling from the non-RT softirq handling (Sect. 3.1). Finally, we mitigate the cache pollution problem and avoid the second priority inversion in a guest OS by outsourcing [9] (Sect. 3.2).

Compared to the RT Preempt Patch Only method, the outsourcing method has the 76% lower standard deviation, 33% lower CPU overhead, and 15% higher throughput. Compared to the dedicated processor method, the outsourcing method has the 63% lower standard deviation and higher total throughput by a factor of 2, and avoids under-utilization of the dedicated processor.

2 The Latency Variance Problems in Vanilla Linux and Two Production RT Methods

In this section, we illustrate the latency variance problems in vanilla Linux and two representative production RT methods with a common mix of an RT service and co-located non-RT network services called NetRT and NetStream (Fig. 1). A NetRT server is an RT network service that receives requests from clients sporadically and replies with response messages to the clients. A NetStream server is a non-RT network service that receives messages continuously from clients in a best effort manner but it does not send response messages. While a NetRT server requires short and predictable response times, a NetStream server desires high throughput.

The NetRT server uses an RT network and the NetStream servers use non-RT networks. In the following, we refer to the Network Interface Cards (NICs) connected to these networks as RT NIC and non-RT NICs, respectively. We assume that the network delay and bandwidth of the RT network are guaranteed by using the methods described by [10–13]. The non-RT networks are best-effort networks.

In this figure, we run a network service in a VM. We allocate a single vCPU to each VM of NetStream, and the vCPU corresponds to a host thread with a normal priority. We allocate two vCPUs to the VM of NetRT. The first vCPU (non-RT vCPU) executes system tasks (e.g. housekeeping tasks) and it corresponds to a host thread with a normal priority. The second vCPU (RT vCPU) executes the NetRT server and it corresponds to a host thread with a high priority. Each VM has a vNIC thread that executes a backend driver of the VM network.

2.1 Interrupt Handling of Network Devices in Vanilla Linux

Linux implements the split interrupt handling model to handle interrupts. Figure 1a shows the interrupt handling in vanilla Linux. Each device driver of a NIC has two interrupt handlers: the *hard IRQ handler* and *softirq handler*. The hard IRQ processes the essential interrupt tasks while interrupts are disabled and the softirq handler processes the remaining interrupt tasks including heavy TCP and bridge processing while interrupts are enabled.

Each CPU has an instance of the softirq mechanism and this instance is shared by multiple device drivers. To ensure cache affinity, the softirq handler of a device driver is executed by the same CPU that receives the IRQ from the device and that executes the hard IRQ handler. The hard IRQ handler of a NIC put the RT softirq handler into the *poll_list*, which is the list of pending softirq handlers in a per-CPU variable.

The interrupt handling in vanilla Linux has a priority inversion problem. Interrupt handlers are executed prior to user processes. For instance, in Fig. 1a, the RT vCPU thread, which is a high-priority user process, can be delayed by the softirq handler of a non-RT NIC.

In this configuration, message copying is performed two times, i.e., once between the host kernel and a guest kernel by a vNIC thread, and another from the guest kernel to a guest user process by a guest kernel. This message copying causes a cache pollution problem. We will discuss about it in Sect. 2.5.

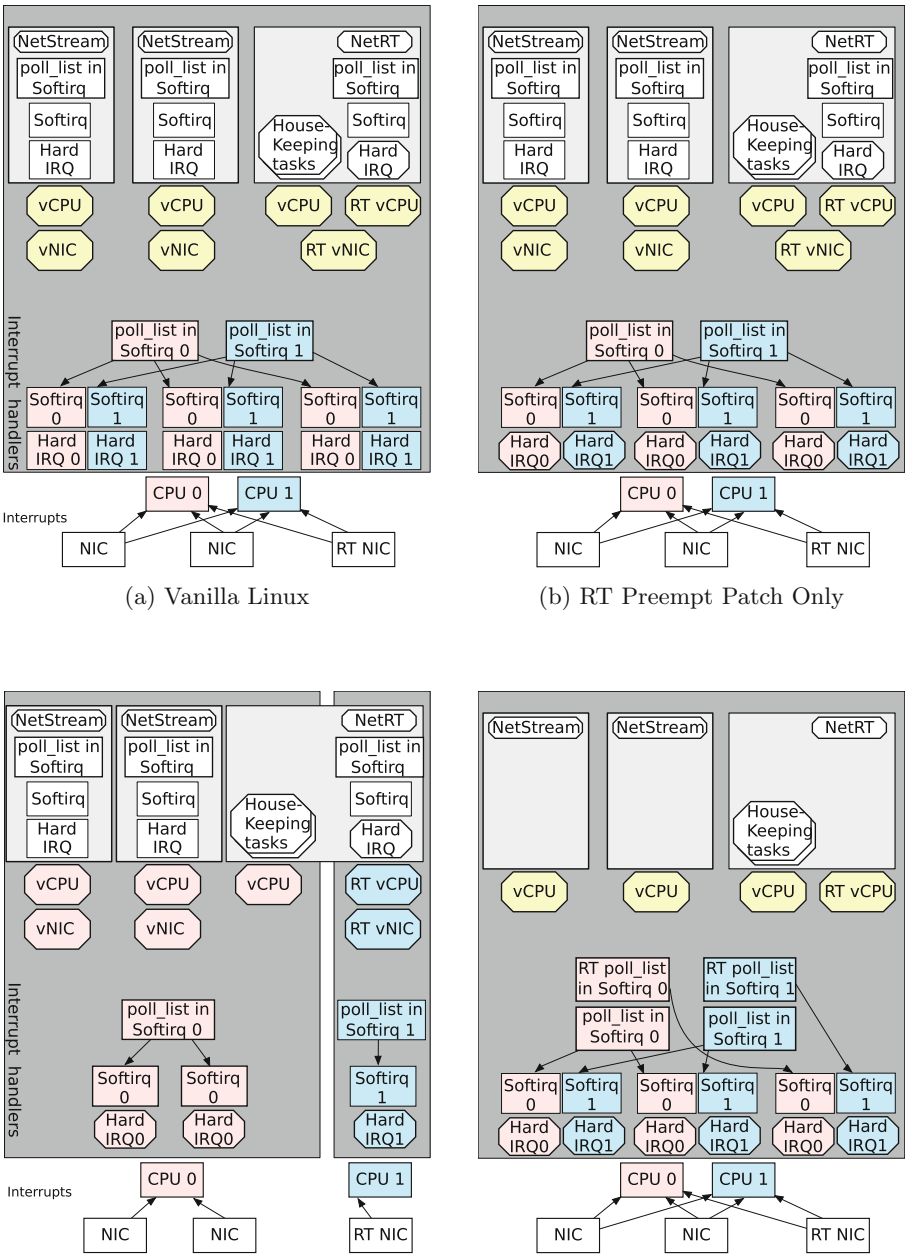
2.2 The RT Preempt Patch Only Method

We can solve the priority inversion problem in Sect. 2.1 by using the RT Preempt patch [8]. We call this method the *RT Preempt Patch Only method*. This patch makes the kernel more preemptible by the following features:

- Hard IRQ handlers are executed as threads (IRQ threads).
- Spin locks are translated into mutexes with a priority inheritance function.

Figure 1b shows that the threads of VMs and interrupt handlers in the RT Preempt Patch Only method. In this figure, each NIC has two IRQ threads for two CPUs. Each IRQ thread is bounded to a CPU.

This method solves the priority inversion problem in Sect. 2.1. Each IRQ thread calls hard and softirq handlers with its own priority. In Fig. 1b, the IRQ



(a) Vanilla Linux

(b) RT Preempt Patch Only

(c) Dedicated processor

(d) Outsourcing

Fig. 1. Configurations of vanilla Linux and the RT methods.

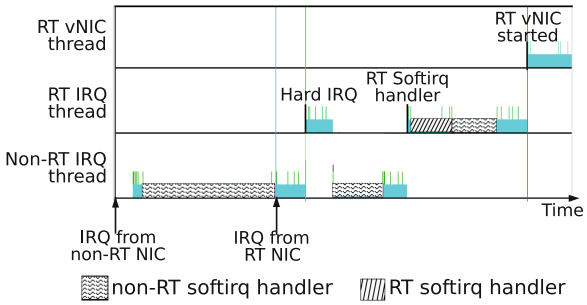


Fig. 2. Priority inversion in interrupt handling of the host OS.

threads of the non-RT NICs do not preempt the threads of the RT VM. This method can yield high achievable CPU utilization because all CPUs execute any vCPU and vNIC threads.

2.3 The Priority Inversion in the Softirq Mechanism of Linux

The RT Preempt patch solves the priority inversion in Sect. 2.1. However, there exists another type of priority inversion in the softirq mechanism of Linux.

Figure 2 shows a trace of the kernel activities using the RT Preempt Patch Only method, where we obtained this plot using KernelShark [14][1]. In this figure, while the CPU was executing the non-RT IRQ thread that called the non-RT softirq handler, the CPU received an interrupt from an RT NIC. The CPU activated the RT IRQ thread, and called the RT hard IRQ handler. The RT hard IRQ handler put the RT softirq handler into the poll_list in a per-CPU variable.

After finishing the RT hard IRQ handler, the RT IRQ thread entered the softirq mechanism. This thread tried to lock the per-CPU variable but it was locked by the non-RT IRQ thread. Therefore, the CPU suspended the RT IRQ thread and executed the non-RT IRQ thread. This thread continued the non-RT softirq handler. At this time, this thread ran with a high priority according to the priority inheritance function. Therefore, the RT IRQ thread had to wait until the non-RT IRQ thread finished. This is a priority inversion.

Next, in Fig. 2, the non-RT softirq handler exceeded the execution quota. Therefore, the non-RT softirq handler was put at the end of the poll_list. Next, the RT IRQ thread obtained the lock and called the RT softirq handler. This handler processed messages from the RT NIC, placed messages into a queue, and activated the RT vNIC thread. Next, the RT IRQ thread also called the non-RT softirq handler with high priority, which created a priority inversion.

[1] The results of KernelShark may include large probe effects.

2.4 Dedicated Processor Method

We can avoid the priority inversion in Sect. 2.3 by dedicating processors to RT threads. This is a popular production method to run RT services in commodity hosted environments [15–18]. We call this method *the dedicated processor method*.

Figure 1c shows that this method allocates a group of RT threads to a dedicated CPU. We call such a CPU an RT-CPU. In this method, an RT NIC injects interrupts to an RT-CPU and a non-RT NIC injects interrupts to a non-RT CPU. Interrupt handling of non-RT NICs do not disturb the execution of the RT threads.

While this method can achieve consistently low latency, it has a drawback. Because RT CPUs do not help to execute non-RT threads, this method yields less achievable CPU utilization.

2.5 Cache Pollution Problem

The CPUs (cores) of a system are explicitly-shared resources and we can control them through priorities of threads and dedication. On the other hand, the Last Level Cache (LLC) is an implicitly-shared resource. When co-located non-RT services pollute the LLC, this can interfere the execution of RT services. For example, in Fig. 1b, when the NetStream servers receive messages, vNIC threads and guest operating systems copy these messages and this copying pollutes the LLC. This changes the response times of the NetRT server.

Note that we cannot avoid this problem using the dedicated processor method. This is because in many CPU architectures, the LLC is shared among CPU cores.

3 Outsourcing Method

Sections 2.2 and 2.4 described two production RT methods, the RT Preempt Patch only method and the dedicated processor method. This section shows our proposed method, *the outsourcing method*.

Figure 1d shows the configuration of the method. This method is an extension of the RT Preempt Patch Only method. It uses the RT Preempt patch and it assigns high priorities to RT threads. This method avoids the priority inversion in Sect. 2.3 by adding a new poll_list for RT services as shown in Fig. 1d. In addition, this method mitigates the cache pollution problem in Sect. 2.5 and avoids the priority inversion in a guest OS by RT socket outsourcing.

3.1 Adding an RT Poll_list for RT Services

In the outsourcing method, we divide the poll_list of the softirq mechanism into two lists.

- The poll_list for non-RT NIC handlers (the non-RT poll_list).
- The poll_list for RT NIC handlers (the RT poll_list).

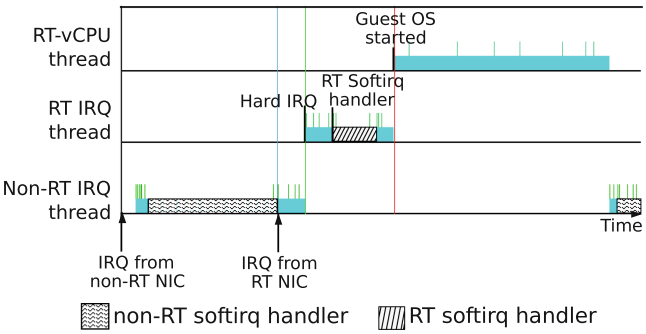


Fig. 3. Separating RT interrupt handling from non-RT interrupt handling.

Figure 3 shows interrupt handling in the outsourcing method. While the CPU was executing the non-RT softirq handler, the CPU received an interrupt from an RT NIC as in Fig. 2. The CPU activated the RT IRQ thread, and called the RT hard IRQ handler. The RT hard IRQ handler placed the RT softirq handler into the RT poll_list.

After finishing the RT hard IRQ handler, the RT IRQ thread entered the softirq mechanism. This thread obtained the lock of the per-CPU variable and called the RT softirq handler. In contrast to the RT Preempt Patch Only method, it called the RT softirq handler and it did not call the non-RT softirq handler because the RT poll_list only contained the RT softirq handler. After the RT IRQ thread finished processing the RT message, it made the CPU available to the RT vCPU thread. In contrast to Fig. 2, there is no priority inversion in Fig. 3.

We implemented the RT poll_list in Linux, which required the changing of 150 lines of code. First, we added the code for the RT poll_list by duplicating that for the base poll_list. Second, we changed the function `napi_schedule_irqoff()`. This function uses the RT poll_list instead of the non-RT poll_list if the IRQ thread is labeled as RT. This allows the reuse of the existing device drivers without modification. We did not change the device driver of the Intel X520 NIC. We added the sysctl parameter `net.core.rtnet_prio` to label the IRQ threads as RT. For example, by calling `sysctl -w net.core.rtnet_prio=47`, an IRQ thread running with a priority equal to or higher than 47 uses the RT poll_list in `napi_schedule_irqoff()`.

3.2 RT Socket Outsourcing

To overcome the latency variance caused by cache pollution, we extend socket outsourcing [9]. 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. The incoming network messages arriving at a guest process are handled by the host network stack.

Socket outsourcing is implemented using VMRPCs [9]. VMRPCs are remote procedure calls for hosted VMs that allow a guest client to invoke a procedure

on a host server. The parameters for this procedure are passed through the shared memory. VMRPCs are synchronous in a similar manner to system calls, so a naive implementation may block a client until the host procedure returns a response message. Conventional socket outsourcing uses virtual interrupts to solve this problem. Therefore, it has the priority inversion problem in the softirq mechanism of a guest OS.

We solve this problem by eliminating interrupt handling from a guest OS. We call this *RT socket outsourcing*. In RT socket outsourcing, the vCPU running an RT server waits for incoming messages in the idle process. The idle process in the guest OS executes the halt instruction and this makes the vCPU thread sleep in the host OS. When new messages arrive in the host OS, the host vCPU thread and the guest idle process are resumed. The idle process checks the event queue and the structure with the states of the sockets in the shared memory, wakes up the receiving processes and goes back to the scheduler. The scheduler executes these processes immediately without interrupt handling. Further, receiving new messages does not interfere with a running RT service. When the RT service is running and a new message arrives, the guest kernel does not handle the new message immediately. The guest kernel handles it when the RT service issues a receive system call or the guest kernel becomes idle.

Message copying is performed two times in current production methods, including the RT Preempt patch method and dedicated processor method, i.e., once between the host kernel and a guest kernel, and another from the guest kernel to a guest user process. By contrast, in socket outsourcing, message copying is performed once from the host kernel to a guest user process. This omission of copying makes the footprint smaller and reduces the cache pollution by non-RT services. This contributes lower latency variance of RT services.

We have implemented RT socket outsourcing as kernel modules. The guest module replaces the socket layer functions with those that perform VMRPCs to the host procedures. We modified the idle process in the guest, which examines the event queue and the socket status structure. A module in the host contains procedures for handling the requests from guest clients.

4 Experimental Evaluation

4.1 Experimental Setup

Figure 4 shows the experimental environment. We used netperf [19] as the NetRT server. Using a remote client, we measured round trip times. We modified the client of netperf, which sent request messages at random intervals ranging from 1 to 10 ms using UDP and received the response messages. We ran iperf [20] in server mode as a NetStream server. The iperf client sent continuous messages using TCP at the maximum speed.

It should be noted that changing the inter-arrival time had a similar effect on the CPU cache as changing the streaming workload. When we made the inter-arrival time shorter, the cache could retain more contents of the NetRT server, which corresponded to making the load lighter. When we made the inter-arrival

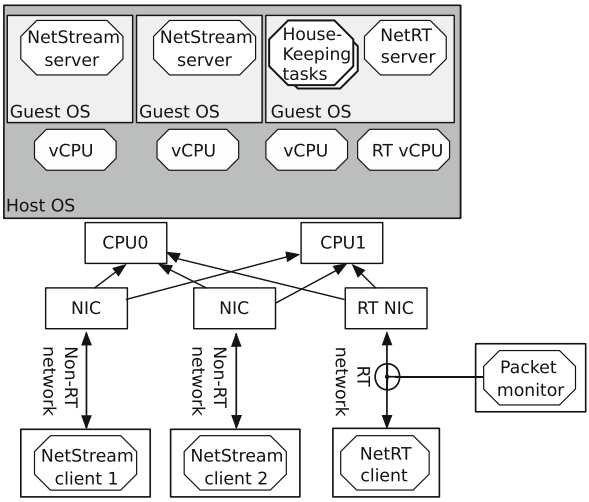


Fig. 4. The experimental environment.

Table 1. Configurations of the machines and their active cores in the experiments.

Machine	CPU/Cache (MB)	Cores	OS
VM host	Intel Core i7-6700K/8	2	Linux 4.1
NetRT client	Intel Core i7-6700K/8	4	Linux 4.1
NetStream client 1	Intel Core i7-3820/10	4	Linux 4.1
NetStream client 2	Intel Core i7-3820/10	4	Linux 4.1
Monitor	Intel Core i7-3820/10	4	Linux 3.16

time longer, the cache could retain fewer contents of the NetRT server, which corresponded to making the load heavier.

We connected the host of the VMs with a single RT network and two non-RT networks, as shown in Fig. 4. These networks comprised 10GBASE-LR optical fiber Ethernet systems. The NICs were Intel X520 Ethernet converged network adapters. We used two non-RT network links to make the CPUs busy on the VM host. When we used a single link, this link became the bottleneck and the CPUs had idle times. We set the maximum transfer unit (MTU) for these networks to 1500 bytes.

We measured the response times of the NetRT server using a hardware monitor, i.e., an Endace DAG10X2-S card [21]. We inserted optical taps into the RT network and directed packets to the Endace DAG card. This card captured and timestamped both the request and response packets at a resolution of 4 ns.

Table 1 shows the specifications of the physical machines used in the experiments. To avoid fluctuations in the frequency of the CPUs, we disabled the

Table 2. NetRT latency comparison between the RT methods (microseconds).

Method	Mean	99th percentile	Standard deviation
No method (vanilla Linux)	123.0	184.9	22.0
RT Preempt Patch Only	91.8	120.5	11.8
Dedicated processor	58.1	80.8	7.7
Outsourcing	32.6	49.5	2.8
Outsourcing (only RT poll_list)	93.2	131.1	11.4
Outsourcing (only RT socket outsourcing)	41.1	64.0	12.2

following processor features: Hyper-threading, TurboBoost and C-States[2]. In addition, we set the `CONFIG_NO_HZ_FULL` option in both the host and guest kernels to reduce the number of clock ticks in the CPU that executed the NetRT server. For the dedicated processor method, we assigned CPU 0 as the non-RT CPU and CPU 1 as the RT CPU. All guest OSes were Linux 4.1.

4.2 Experimental Results

Figure 5 shows the response times of the NetRT server, and Table 2 shows their statistical values (the mean, 99th percentile, and standard deviation (SD)). Figure 6 shows the throughputs and Fig. 7 shows the achievable CPU utilization.

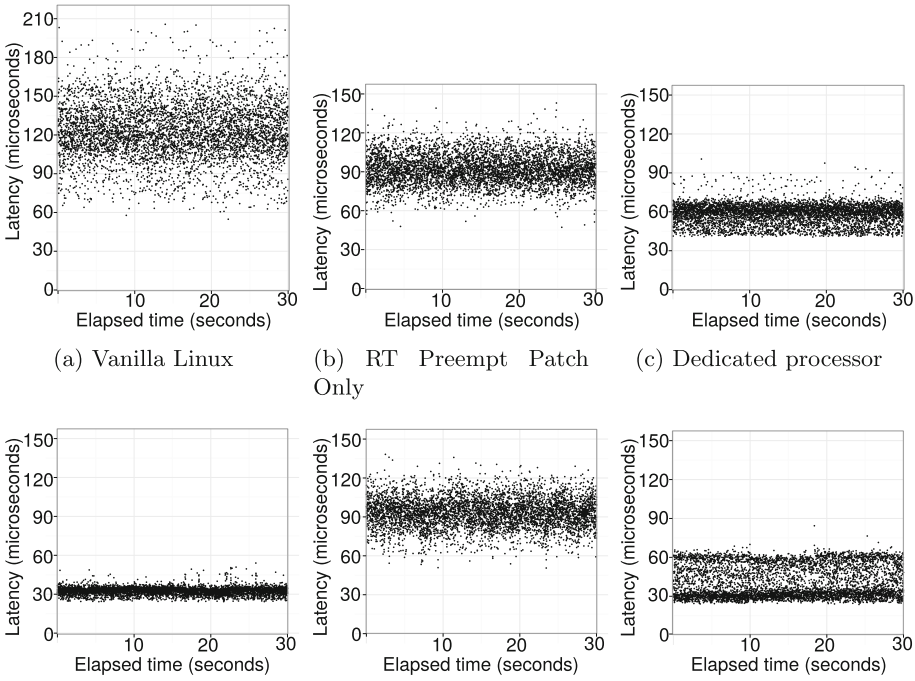
Figure 5a shows that in vanilla Linux, the NetRT server had high latency variance. On the other hand, the execution of NetStream servers presented a high performance as shown in Fig. 6, and NetStream servers got a throughput of 18.8 Gbps over an aggregated link capacity of 20 Gbps. The activities of the NetRT server and the NetStream servers did not consume all the CPU resources, as shown in Fig. 7.

Figure 5b shows that the RT Preempt Patch Only method improved the system realtimeness compared with vanilla Linux. Despite this improvement, the NetStream servers interfered with the response times of the NetRT server. The CPUs reached their maximum capacities in this method, which limited the volume of data received by the NetStream servers. Consequently, the NetStream servers had a throughput of 16.0 Gbps.

Figure 5c shows the response times obtained using the dedicated processor method. This method reduced the latency variability. However, the dedicated processor method could only use 50% of the CPU resources to execute the NetStream servers, which limited the total throughput to 8.6 Gbps.

Figure 5d shows the response times using the outsourcing method that comprises the two techniques: adding an RT poll_list (Sect. 3.1) and RT socket outsourcing (Sect. 3.2). This method had the lowest latency variability among the

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



(a) Vanilla Linux (b) RT Preempt Patch Only (c) Dedicated processor

(d) Outsourcing (using both techniques) (e) Outsourcing (only adding an RT poll_list). (f) Outsourcing (only RT socket outsourcing).

Fig. 5. Distribution of the NetRT server response times.

three methods. In addition, the outsourcing method maintained a high throughput of 18.8 Gbps with the lowest CPU consumption.

In summary, compared to the RT Preempt Patch Only method, the outsourcing method has the 76% lower standard deviation, 15% higher throughput, and 33% lower CPU overhead. Compared to the dedicated processor method, the outsourcing method has the 63% lower standard deviation and higher total throughput by a factor of 2, and avoids under-utilization of the dedicated processor.

We performed the experiment by enabling one of two techniques: adding an RT poll_list and RT socket outsourcing. Figure 5e and f show the results. When we enabled only one of the two techniques, we obtained large variability in the response times.

4.3 Application Benchmarks

In Sect. 4.2, we compared the three RT methods using netperf as a NetRT server. In this section, we compare these methods using two time-sensitive applications as NetRT servers. We used the same experimental environment and configurations described in Sect. 4.1.

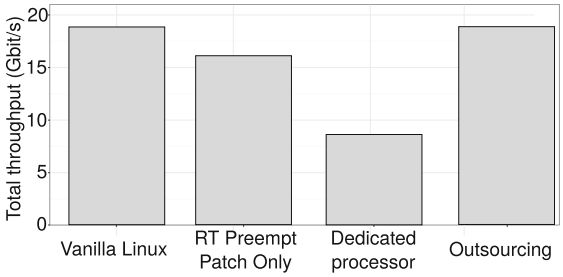


Fig. 6. Total throughput.

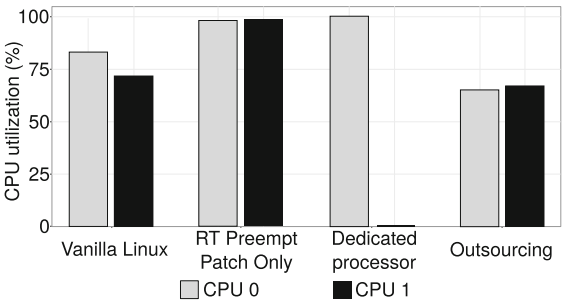


Fig. 7. Achievable CPU utilization.

A Voice over IP Server (VoIP). We measured the forward delays of a VoIP server that used the Session Initiation Protocol (SIP). We measured the impact of the NetStream servers on the activity of the NetRT server with various requesting rates. In this experiment, we ran Kamailio [22], a VoIP server, as the NetRT server. In a remote machine, we ran a SIPp [23] instance as a user agent client (UAC) and another instance as a user agent server (UAS). The VoIP server relayed messages between the UAC and the UAS.

We measured the forward delays between the message the VoIP server received and the message the VoIP server sent in SIP calls. In a single SIP call, the server forwarded six messages. The UAC first sent an INVITE message to the server, the server replied with a TRYING message and forwarded the INVITE message to the UAS. Next, the server forwarded a RINGING and OK message from the UAS to the UAC. Next, the server forwarded an ACK and BYE message from the UAC to the UAS. Finally, the server forwarded an OK message from the UAS to the UAC. We measured these forward delays using the hardware monitor, the Endace DAG card. We modified SIPp to make calls at random rates ranging from 17 to 167 calls per second. This means that the server forwarded 100 to 1000 messages in a second.

Figure 8 shows the 50^{th}, 99^{th} and 99.9^{th} percentiles of the NetRT response times. The response times with the outsourcing method had lower tail latencies than those with the RT Preempt Patch Only method and the dedicated processor

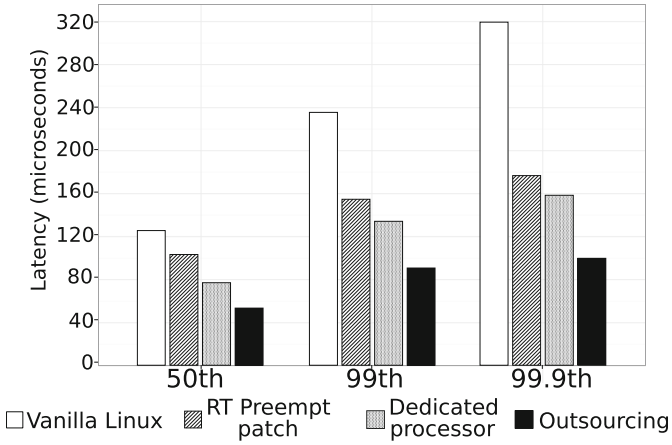


Fig. 8. The forward delays of a voice over IP server (Kamailio).

method. For example, in the 99th percentiles, the outsourcing method had 41% lower latency compared with the RT preempt Patch Only method and 32% lower latency compared with the dedicated processor method.

Memcached. We performed another application experiment using Memcached [24] as a NetRT server. Memcached is a distributed caching server that stores key-value pairs. The NetRT server received requests from a remote client called memaslap [25]. We measured the response times using random request intervals ranging from 100 to 1000 requests per second with the hardware monitor, the Endace DAG card. Memaslap sent GET/SET requests at a ratio of 9:1. The size of a key was 64 bytes and the size of the value was 1024 bytes. We ran the same NetStream servers employed in the previous experiments.

Figure 9 presents the 50th, 99th, and 99.9th percentiles of the response times of the GET requests. Similar to the previous experiment, the outsourcing method obtained better results than the RT Preempt Patch Only and the dedicated processor methods. In the 99th percentiles, the outsourcing method had 59% lower latency compared with the RT preempt Patch Only method and 54% lower latency compared with the dedicated processor method.

5 Related Work

The RT Preempt Patch Only and dedicated processor methods [8,15,16,18] are among the favorite choices in production-use operating systems that support RT services. Classic proposals for adding real-time support to commodity operating systems include RTLinux [26], Time-Sensitive Linux (TSL) [27], and Xenomai [28]. Although these proposed systems have been shown to be effective in reducing latency variance, the results are affected by the continuous evolution of the

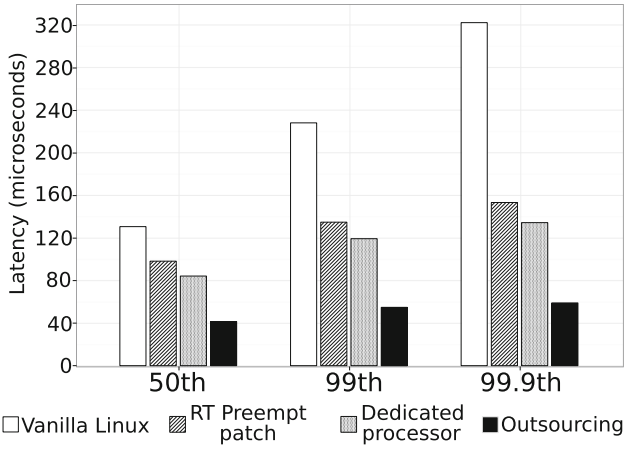


Fig. 9. The response times of Memcached.

underlying OS code. Examples of new sources of variance described in Sects. 2.3 and 2.5 include priority inversion and cache pollution.

Other proposals to address the priority inversion problem in network processing include lazy receiver processing (LRP) [29], which postpones the interrupt handling process until execution of the receiver task, and prioritized interrupt handling [30] for asynchronous transfer mode networks. More generally, some user space I/O frameworks [31–33] employ a polling mode to avoid latencies caused by interrupt handling and allow applications to send and receive packets directly from the DMA buffers of a NIC. Other alternatives to improve network throughput include polling threads [31], Data Plane Development Kit (DPDK)/vhostuser [32] and Netmap [33]. These design and implementation alternatives have varied trade-offs in throughput, latency, variance in latency, and achievable CPU utilization, as well as implementation difficulty in commodity systems such as Linux and KVM.

The study of software-based methods in this paper complements the advanced hardware assist techniques to improve I/O performance in VM environments [10–12,34–36]. Concrete examples include: Exit-Less Interrupts (ELI) [34], Efficient and Scalable Paravirtual I/O System (ELVIS) [35], and Direct Interrupt Delivery (DID) [36], which employ advanced hardware features, such as Single Root I/O Virtualization (SR-IOV) and Advanced Programmable Interrupt Controller virtualization (APICv) by Intel, Advanced Virtual Interrupt Controller (AVIC) by AMD, and Virtual Generic Interrupt Controller (VGIC) by ARM. These new hardware features are able to bypass some of the software layers in a consolidated environment that includes a guest OS, the host OS, and the VMM. The combination of software techniques such as outsourcing with advanced hardware assist is another interesting area of future research.

Previous outsourcing and similar techniques focus on improving throughput [9,37–39]. This is the first paper that uses outsourcing for reducing latency and variance in latency.

6 Conclusion

In this study, we have proposed the outsourcing method of real-time network services in KVM-based commodity hosted environments, and evaluated our method by comparing with two production methods, the RT Preempt Patch Only method and the dedicated processor method.

First, we found that the RT Preempt Patch Only method is able to reduce and remove the sources of latency variance in interrupt handling, primarily due to the first priority inversion problem between RT user processes and non-RT interrupt handling. However, the second priority inversion problem in the softirq mechanism of Linux remains. The dedicated processor method dedicates an exclusive processor for RT threads, including softirq, thus removing all thread-related priority inversion problems. Its drawback is that the low utilization of the dedicated processor can significantly reduce the total achievable throughput. The main contribution of the paper is the outsourcing method, which is implemented with two modest modifications to Linux (in addition to the RT Preempt patch). The first modification explicitly separates RT softirq handling from non-RT softirq handling, removing the second priority inversion problem. The second modification outsources the processing of network services from the guest OS to the host OS, thereby mitigating the cache pollution problem and avoiding the second priority inversion in a guest OS.

Compared to the RT Preempt Patch Only method, the outsourcing method has the 76% lower standard deviation, 15% higher throughput, and 33% lower CPU overhead. Compared to the dedicated processor method, the outsourcing method has the 63% lower standard deviation, higher total throughput by a factor of 2, and avoids under-utilization of the dedicated processor.

Acknowledgments. This work was partially supported by JSPS KAKENHI Grant Number 25540022 and 16K12410.

References

1. Prevot, T.: NextGen technologies for mid-term and far-term air traffic control operations. In: IEEE/AIAA 28th Digital Avionics Systems Conference, pp. 2.A.4-1–2.A.4-16 (2009)
2. Chen, I.R., Guo, J., Tsai, J.J.P.: Trust as a service for SOA-based IoT systems. Serv. Trans. Internet Things (STIOT) **1**(1), 43–52 (2017)
3. Page, A., Hijazi, S., Askan, D., Kantarci, B., Soyata, T.: Support systems for health monitoring using Internet-of-Things driven data acquisition. Int. J. Serv. Comput. (IJSC) **4**(4), 18–34 (2016)

4. Chang, S.F., Chang, J.C., Lin, K.H., Yu, B., Lee, Y.C., Tsai, S.B., Zhou, J., Wu, C., Yan, Z.C.: Measuring the service quality of E-Commerce and competitive strategies. Int. J. Web Serv. Res. (IJWSR) **11**(3), 96–115 (2014)
5. LiDuan, Y., Chen, J.: Event-driven SOA for IoT services. Int. J. Serv. Comput. (IJSC) **2**(2), 30–43 (2014)
6. Sun, Y., Qiao, X., Tan, W., Cheng, B., Shi, R., Chen, J.: A low-delay, light-weight publish/subscribe architecture for delay-sensitive IoT services. Int. J. Web Serv. Res. (IJWSR) **10**(3), 60–81 (2013)
7. dmoz.org: List of real-time operating systems. https://dmoztools.net/Computers/Software/Operating_Systems/Realtime/. Accessed 1 Nov 2017
8. Rostedt, S., Hart, D.V.: Internals of the RT patch. In: Proceedings of the Linux Symposium, pp. 161–172 (2007)
9. Eiraku, H., Shinjo, Y., Pu, C., Koh, Y., Kato, K.: Fast networking with socket-outsourcing in hosted virtual machine environments. In: Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), pp. 310–317 (2009)
10. Jang, K., Sherry, J., Ballani, H., Moncaster, T.: Silo: predictable message latency in the cloud. In: Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM 2015), pp. 435–448 (2015)
11. Chowdhury, M., Liu, Z., Ghodsi, A., Stoica, I.: HUG: multi-resource fairness for correlated and elastic demands. In: 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI), pp. 407–424 (2016)
12. Xiong, P., Hacigumus, H., Naughton, J.F.: A software-defined networking based approach for performance management of analytical queries on distributed data stores. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 955–966 (2014)
13. Werner, C., Buschmann, C., Jäcker, T., Fischer, S.: Bandwidth and latency considerations for efficient SOAP messaging. Int. J. Web Serv. Res. **3**(1), 49–67 (2006)
14. KernekShark: KernelShark - a front end reader of trace-cmd (2017). http://rostedt.homelinux.com/kernelshark/. Accessed 27 Oct 2017
15. Red Hat Inc.: Enterprise Linux for Real Time. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux_for_Real_Time/7/html/Installation_Guide. Accessed 13 May 2017
16. Christofferson, M.: 4 ways to improve performance in embedded Linux systems. Korea Linux Forum (2013)
17. Crespo, A., Ripoll, I., Masmano, M.: Partitioned embedded architecture based on hypervisor: the XtratuM approach. In: Proceedings of the IEEE European Dependable Computing Conference (EDCC), pp. 67–72 (2010)
18. van Riel, R.: Real-time KVM from the ground up. KVM Forum, August 2015
19. Jones, R.: Netperf (1996). https://hewlettpackard.github.io/netperf/. Accessed 27 May 2017
20. Tirumala, A., Qin, F., Dugan, J., Ferguson, J., Gibbs, K.: iPerf: the TCP/UDP bandwidth measurement tool (2005). http://iperf.sourceforge.net 20 Apr 2017
21. Endace Technology Limited: DAG10X2-S datasheet (2015). https://www.endace.com/dag-10x2-s-datasheet.pdf. Accessed 12 June 2017
22. The Kamailio SIP Server Project: Kamailio SIP server. https://www.kamailio.org/w/. Accessed 12 May 2017
23. Gayraud, R., Jacques, O.: SIPp benchmark tool. http://sipp.sourceforge.net/. Accessed 23 Apr 2017
24. Memcached: Memcached - a distributed memory object caching system (2015). https://memcached.org/. Accessed 3 July 2017

25. Aker, B.: Libmemcached-Memaslap. http://libmemcached.org/libMemcached.html. Accessed 1 Nov 2017
26. Barabanov, M., Yodaiken, V.: Introducing real-time Linux. Linux J. **34** 9 p. (1997)
27. Goel, A., Abeni, L., Krasic, C., Snow, J., Walpole, J.: Supporting time-sensitive applications on a commodity OS. In: the USENIX 5th Symposium on Operating Systems Design and implementation, pp. 165–180 (2002)
28. Gerum, P.: Xenomai - Implementing a RTOS emulation framework on GNU/Linux. http://www.xenomai.org/documentation/xenomai-2.5/pdf/xenomai.pdf. Accessed 4 Nov 2017
29. Druschel, P., Banga, G.: Lazy receiver processing (LRP): a network subsystem architecture for server systems. In: the 2nd USENIX Symposium on Operating Systems Design and Implementation, pp. 261–275 (1996)
30. Kuhns, F., Schmidt, D.C., Levine, D.L.: The design and performance of a real-time I/O subsystem. In: Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium, pp. 154–163 (1999)
31. Liu, J., Abali, B.: Virtualization polling engine (VPE): using dedicated CPU cores to accelerate I/O virtualization. In: The ACM 23rd International Conference on Supercomputing, pp. 225–234 (2009)
32. Intel Corporation: DPDK: Data Plane Development Kit. http://dpdk.org/. Accessed 28 Sept 2017
33. Rizzo, L.: Netmap: a novel framework for fast packet I/O. In: Proceedings of the 2012 USENIX Conference on Annual Technical Conference, p. 9 (2012)
34. Gordon, A., Amit, N., Har'El, N., Ben-Yehuda, M., Landau, A., Schuster, A., Tsafrir, D.: 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, pp. 411–422 (2012)
35. Gordon, A., Har'El, N., Landau, A., Ben-Yehuda, M., Traeger, A.: Towards exitless and efficient paravirtual I/O. In: Proceedings of the 5th ACM Annual International Systems and Storage Conference (SYSTOR), pp. 1–6 (2012)
36. Tu, C.C., Ferdman, M., Lee, C., Chiueh, T.: A comprehensive implementation and evaluation of direct interrupt delivery. In: Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), pp. 1–15 (2015)
37. Gamage, S., Kompella, R.R., Xu, D., Kangarlou, A.: Protocol responsibility offloading to improve TCP throughput in virtualized environments. ACM Trans. Comput. Syst. (TOCS) **31**(3), 1–34 (2013)
38. Nordal, A., Kvalnes, Å., Johansen, D.: Paravirtualizing TCP. In: Proceedings of the 6th ACM International Workshop on Virtualization Technologies in Distributed Computing Date (VTDC), pp. 3–10 (2012)
39. Lin, Q., Qi, Z., Wu, J., Dong, Y., Guan, H.: Optimizing virtual machines using hybrid virtualization. Elsevier J. Syst. Softw. **85**(11), 2593–2603 (2012)