applied sciences

MDPI

Article

NetAP: Adaptive Polling Technique for Network Packet Processing in Virtualized Environments

Hyunchan Park [1,2], Juyong Seong [1], Munkyu Lee [3], Kyungwoon Lee [4,*] and Cheol-Ho Hong [3,*]

1 Division of Computer Science and Engineering, Jeonbuk National University, Jeonju 54896, Korea; hyunchan.park@jbnu.ac.kr (H.P.); wndyd9557@jbnu.ac.kr (J.S.)
2 Research Center for Artificial Intelligence Technology, Jeonbuk National University, Jeonju 54896, Korea
3 School of Electrical and Electronics Engineering, Chung-Ang University, Seoul 06974, Korea; dse112@cau.ac.kr
4 College of Informatics, Korea University, Seoul 02841, Korea
* Correspondence: kwlee@os.korea.ac.kr (K.L.); cheolhohong@cau.ac.kr (C.-H.H.)

Received: 19 June 2020; Accepted: 27 July 2020 ; Published: 29 July 2020

Abstract: In cloud systems, computing resources, such as the CPU, memory, network, and storage devices, are virtualized and shared by multiple users. In recent decades, methods to virtualize these resources efficiently have been intensively studied. Nevertheless, the current virtualization techniques cannot achieve effective I/O virtualization when packets are transferred between a virtual machine and a host system. For example, VirtIO, which is a network device driver for KVM-based virtualization, adopts an interrupt-based packet-delivery mechanism, and incurs frequent switch overheads between the virtual machine and the host system. Therefore, VirtIO wastes valuable CPU resources and decreases network performance. To address this limitation, this paper proposes an adaptive polling-based network I/O processing technique, called NetAP, for virtualized environments. NetAP processes network requests via a periodical polling-based mechanism. For this purpose, NetAP adopts the golden-section search algorithm to determine the near-optimal polling interval for various workloads with different characteristics. We implement NetAP in a Linux kernel and evaluated it with up to six virtual machines. The evaluation results show that NetAP can improve the network performance of virtual machines by up to 31.16%, while only using 32.92% of the host CPU time used by VirtIO for packet processing.

Keywords: cloud computing; virtualization; I/O processing; adaptive polling

1. Introduction

Artificial intelligence, cloud computing, big data, and the Internet of Things are the main technologies of the fourth industrial revolution. The importance of these technologies is emerging as an international issue that requires significant involvement by academia and industry. Cloud computing provides the main infrastructure of the fourth industrial revolution; it lays the foundation for other major technologies. The computing environment of enterprises and individuals is rapidly moving to the cloud. Many companies already have their own cloud infrastructures, and others are using public cloud platforms, such as Amazon EC2. Moreover, several artificial intelligence services, big data processing, and Internet of Things applications are currently performed on the cloud.

Virtualization is a key technology that enables and sustains cloud computing. Through virtualization, computing resources, such as the CPU, memory, storage, and network devices, of multiple physical nodes are consolidated and managed. Virtualization provides resources that are required by users in the form of virtual machines (VMs) or containers. However, because the resources are virtualized, applications running in cloud computing experience performance deterioration,

in contrast with applications executed in a native environment. The virtualization overhead for CPUs has been addressed with some success, but the overhead for I/O devices, such as network and storage devices, remains. This overhead is a major bottleneck for overall system performance [1,2].

The efficient virtualization of I/O devices is a critical factor in cloud computing. In a cloud system, many users share a small number of I/O devices. If the virtualization overhead increases in proportion to the number of users, the efficiency of the entire system decreases. Cloud computing service providers then need to install additional equipment to meet user demand. This situation leads to a decrease in competitiveness in cloud computing with an increase in the overall cost.

Another issue in I/O virtualization is that the virtualization overhead may appear in the form of CPU consumption. The current virtualization techniques are heavily supported by hardware, but the role of processing I/O devices is still often handled by software. For example, Linux incorporates a kernel-based virtual machine (KVM) virtualization software and adopts an I/O driver for virtualization, called VirtIO [3,4]. VirtIO is responsible for processing requests and responses to and from I/O devices. Therefore, if VirtIO inefficiently uses the CPU for processing I/O operations, it results in additional cost by wasting CPU resources that might otherwise be provided to users in the cloud system.

In this study, we implement NetAP, a novel network I/O processing method for virtualized systems in KVM-virtualized environments. NetAP introduces a periodic polling technique in VirtIO in order to maximize the use of network and CPU resources. The existing VirtIO processes I/O requests in a batch manner when the number of requests is above a certain threshold. Upon meeting the condition, the VM generates a virtual interrupt (vIRQ) to notify VirtIO to process the requests. However, this mechanism causes expensive VM-Exit and VM-Entry operations, which decreases the I/O performance [5,6]. The periodic polling technique of NetAP regularly monitors the I/O request buffers and handles the network requests in a timely manner. This mechanism prevents vIRQs from the VM, because the requests in the buffer are processed before the number of requests reaches the threshold.

NetAP uses the golden-section search algorithm to dynamically find the near-optimal polling interval for various workloads. The golden-section search technique helps NetAP to find the maximum or minimum value within a given range of a unimodal function. The performance evaluation results of NetAP show that it successfully finds a near-optimal polling interval for various scenarios consisting of multiple VMs. NetAP improves the network performance by up to 31.16%, while only using 32.93% of the CPU time that the original VirtIO used for packet processing in the host system.

Our contributions can be summarized as follows:

- We provide a new I/O processing mechanism that is based on adaptive periodical polling; this method uses the network and CPU more efficiently than traditional interrupt-based mechanisms in virtualized environments.
- We provide a fast and accurate adaptation technique that is based on the golden-section search algorithm. Our evaluation demonstrates that the adaptation technique successfully finds a near-optimal polling interval that maximizes network and CPU utilization for various workloads.

This paper is organized, as follows. In Section 2, we provide background information about VirtIO and describes the related work. Section 3 presents the design of NetAP, including the new polling-based mechanism for I/O processing and the adaptation mechanism that is based on the golden-section searching algorithm. The experimental results are presented in Section 4. Section 5 presents our conclusions.

2. Background and Motivation

2.1. VirtIO

VirtIO modules are para-virtualized drivers for virtualized systems based on KVM; such modules are widely used for network and block devices in KVM. Figure 1 shows the structure of VirtIO with request and response flows. When a user application running in a guest OS requests a packet

transmission via a system call, the network stack of the guest OS prepares a packet for transmission and delivers the packet to the VirtIO front-end device driver, which is a virtual network device. The front-end driver uses shared memory for data communication between the VM and the host; the network packets are buffered into the queue in the shared memory region. A vIRQ is then generated in order to notify the host OS that packets are ready for transmission. The vIRQ is generated when the number of packets in the queue exceeds a certain threshold, rather than generating a vIRQ for each packet.

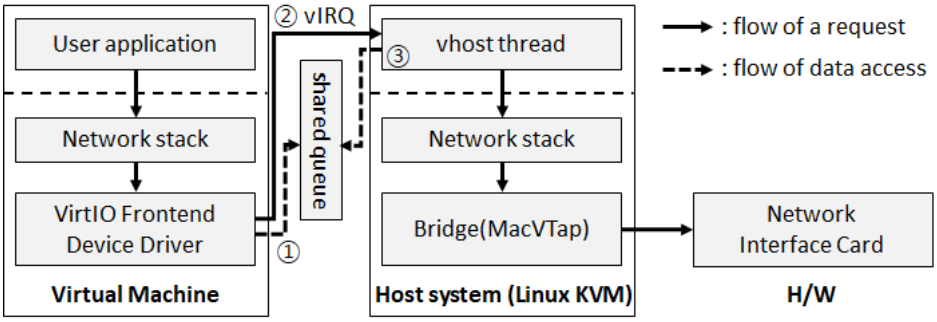


Figure 1. VirtIO structure with request and response flows.

Upon receiving the vIRQ, the host OS schedules the virtual host (vHost) user thread to deliver the packet fetched from the shared queue to the network stack of the host in kernel space. The vHost thread adopts a combined interrupt–polling method, which combines the interrupt and polling mechanisms. After the first vIRQ interrupt, the vHost thread fetches consecutive packets from the shared queue with a polling mechanism and then delivers them to the network stack. The fetched packets are sent to the MacVTap bridge, which is a virtual interface that transmits packets through the physical device. Packet reception is similarly performed in the opposite direction.

The main overhead occurs in the process of requesting packets from the VM to the host. The VM notifies the host system, using a vIRQ, that the packet to be processed is ready in the shared memory. The host system then receives the interrupt and wakes up the sleeping vHost thread to perform packet processing. This leads to frequent VM-Exit and VM-Entry operations to transfer the control between the VM and KVM, which leads to a substantial virtualization overhead [5,6].

2.2. Related Work

2.2.1. Interrupt-Based and Polling-Based I/O Processing Techniques

Interrupt-based and polling-based techniques in I/O processing have the advantages and disadvantages of complementary relationships; furthermore, such techniques are used selectively depending on the workload characteristics. For example, G. Lettieri et al. analyzed the producer-consumer model in I/O processing in a virtualized environment [7]. They compared the busy-waiting method and the sleeping method. The busy-waiting method uses a polling-based technique that continuously processes network packets on a dedicated CPU core. The sleeping method periodically executes polling operations in order to process the stored network packets in a buffer [7]. As a result, it was found that it is efficient to use different I/O processing methods depending on the characteristics of the workload. Specifically, they found that the I/O processing efficiency varied, depending on the period of the polling operations in the sleeping method, which motivated our research.

Recently, high-speed I/O devices, such as 10 GbE and NVMe SSDs, have emerged. As interrupts occur at very high speeds in these devices, the overhead of interrupt delivery becomes a bottleneck in the I/O processing path [8]. To alleviate the overhead, busy-waiting polling techniques are used in many systems that focus on high-speed network processing [9,10]. The busy-waiting techniques

successfully improve network performance by processing incoming network packets continuously. However, more than one CPU core should be dedicated to the busy-waiting workers for packet processing. The workers occupy the dedicated CPU cores even though there are no incoming packets. This leads to inefficient CPU management, because other processes cannot utilize the dedicated CPU cores. This paper proposes an adaptive polling technique that occupies CPU cores when there are incoming packets in order to overcome the inefficient CPU management of busy-waiting techniques.

There have been similar approaches to enhancing the busy-waiting techniques for storage devices. For example, T.Y Kim et al. proposed the use of a periodic polling technique for NVMe SSDs [11]. However, their evaluation results show that the periodic polling technique is less efficient than interrupt handling. This is because the storage used in the research only handles 400 K requests per second. On the other hand, this paper focuses on a 10 GbE network device handling 14,880 K requests per second for 64 B packets, which is approximately 30 times faster than the storage device. Even for 1500-B packets, there are 812 K requests in network devices, which imposes more interrupt-processing overhead than in the case of the storage device. Additionally, previous engineering work suggests adaptive polling, which is similar to NetAP, for block devices on KVM environments [12]. They limit the time for busy-waiting polling algorithm and increase or decrease the polling duration through a simple condition loop. When an I/O event occurs, the polling starts for a certain duration, for example, 32 us (microseconds). If additional I/O events occur, the polling duration increases. Otherwise, the polling duration decreases, which is very simple optimization. The simple optimization does not guarantee finding the optimal time for polling and lacks the mathematical proof and runtime analysis, which is different from the golden-section search algorithm that we adopt. NetAP determines the near-optimal polling interval while using the golden-section search algorithm, which is much more efficient and effective than the simple optimization.

2.2.2. Efficient I/O Processing in a Virtualized Environment

Research for improving the I/O processing performance in virtualized environments can be categorized into two types: one is to use additional hardware devices to increase I/O processing performance and the other is to reduce the data-copying overhead in I/O processing.

First, single root I/O virtualization (SR-IOV) is a technique that allows VMs to directly access physical network devices without the intervention of the host system [13–15]. Allowing the VMs to process I/O requests directly increases the I/O processing performance significantly. However, in SR-IOV, I/O processing is based on interrupts; therefore, it incurs a large overhead when the interrupts occur at high speeds. To reduce the overhead, there have been attempts to minimize the VM-Exit and VM-Entry operations using software methods [16] or hardware methods [17]. Furthermore, previous studies propose assigning specific CPU cores to dedicated vHost threads to mitigate the interrupt handling overhead [18,19]. However, assigning CPU cores for I/O processing degrades the CPU utilization at moderate I/O processing rates [20]. Thus, in existing cloud environments, SR-IOV is only used for users who require very high or stable network performance at high prices.

The second approach is to reduce the data-copying overhead. To this end, one study proposed a zero-copy technique that eliminates memory copying in the packet forwarding path to speed up processing and reduce CPU usage [21]. Complementary with the proposed method, it would be more efficient if the adaptive polling of NetAP and a zero-copy method were simultaneously used.

2.2.3. Research for Optimizing the Polling Interval

Many previous studies have tried to overcome the limitations of interrupt-based and polling-based techniques. Some studies [22] suggest changing the period of polling operations by creating and evaluating a simple cyclic transformation equation through modeling and simulation. They focus on general network processing, rather than on I/O processing in virtualized environments. Hence, the results of previous studies [22–24] are difficult to apply to virtualized environments. Furthermore,

in a real system environment, there are other considerations, such as the period of performance monitoring and the change in the packet-processing speed of the VM, which are not included in the previous studies. Therefore, more in-depth research is necessary in order to optimize the polling intervals in virtual environments. This study provides a search method to optimize the polling interval and a detailed analysis of the period of performance monitoring, which is essential for the search method.

3. Design

This section describes the design goals, the periodic polling technique, and the algorithm for determining the optimal polling interval of NetAP.

3.1. Design Goals

The design goals of NetAP include improving deployability and obtaining the optimal polling interval for each VM, which are explained, as follows:

First, to improve deployability, NetAP only modifies the vHost thread of the host system. If we modify the front-end device driver of each guest OS, the user may experience an inconvenience from re-compiling the front-end driver and re-installing the driver module into the guest OS. Therefore, by modifying only the vHost thread of the host OS, we preserve the interface with an unmodified front-end driver.

Second, NetAP tries to obtain the optimal interval for each VM to reflect its individual workload characteristics. A vHost thread is dedicated to its own VM and it fetches network requests from its dedicated shared queue. Because each VM generates and consumes packets at a different speed, each vHost thread needs to have different polling intervals. Therefore, we apply a novel algorithm to determine the polling interval for each vHost thread.

3.2. Periodical Polling for Network Packet Processing

NetAP develops a periodic polling technique in the vHost thread to maximize the utilization of network and CPU resources. The modified vHost thread in NetAP periodically polls the shared queue instead of waiting for a vIRQ from the front-end device driver. At each polling interval, the vHost thread is scheduled, and it processes packets sent from the guest OS. After the vHost thread completes packet processing, it sleeps until the next polling interval. During this sleep time, the guest OS generates packets, and they are buffered into the shared queue. At the next interval, the vHost thread wakes up and processes the buffered packets in the shared queue.

The main difference between NetAP and the original VirtIO is that NetAP frequently monitors the shared queue and proactively handles the buffered packets before a vIRQ is generated from the front-end driver. In original VirtIO, the front-end driver produces packets and inserts them into the shared queue. When the number of buffered packets in the queue is more than a certain threshold, a vIRQ is delivered to the vHost thread to wake up the thread. However, NetAP processes the buffered packets at an appropriate rate with periodic polling before the number of packets exceeds the threshold. Therefore, the front-end driver does not generate vIRQs. This mechanism can prevent expensive VM-Exit and VM-Entry operations and, therefore, improve network performance with efficient CPU use.

3.3. Periodic Polling Algorithm

Algorithm 1 details the periodic polling technique. The core function is *the_packet_handler_of_a_host_system*(), which processes packets in a continuous loop. When the loop is executed once, it processes the packets loaded in the RX and TX queues (line 21 and 22), sleeps for an *interval* (line 24), and starts the next loop. Until it sleeps, interrupts are disabled for packet arrivals (line 8 and 23). A *round* is a time unit for observing performance. To adapt NetAP for a given workload, we change the polling interval for each round according to the observed

performance for the previous round. Furthermore, *round_length* is the predetermined length of a round (e.g., one second). Line 14 sets the start time of the next round by adding *round_length* to *current_time*, and line 9 checks the time. At the start of a new round, the *interval* is updated by running the *golden_section_searching*() function on line 10. The parameters are the number of the current round, the size of the packet processed in the round (*bytes*), and the current interval. The function is explained in detail in the next section.

The algorithm also provides a function to restart adaptation in response to characteristic changes in the workload after adaptation was formerly completed. Line 11 checks whether the size of transferred data changes greater than the predefined ratio *MinChange*, when the interval is no longer updated. Line 17 checks whether the current round exceeds the predefined maximum number of rounds. If one of the two conditions is satisfied, then adaptation is started again by setting the round number to 0. This allows for NetAP to keep tracking the appropriate interval continuously against the changes of workload characteristics within the running VM.

3.4. Adaptation Using Golden-Section Search

We use the golden-section search algorithm to find the near-optimal polling interval for various workloads dynamically. The golden-section search technique searches the maximum or minimum value within a given range in a unimodal function [25]. The method to find the maximum value for the target function $f(x)$ is as follows: First, we set two x-coordinates x_l and x_u to search. We assume that the maximum value is between x_l and x_u. Second, we choose x_1, where $x_l < x_1 < x_u$ and the $(x_u - x_1)/(x_u - x_l)$ is R, a reciprocal of the golden ratio ($R \approx 0.618$). These three x-coordinates are golden-section triplets. Third, we set the new x-coordinate, x_2, where $x_l < x_2 < x_u$ and the $(x_2 - x_l)/(x_u - x_l)$ is R. Subsequently, we evaluate the $f(x_l)$, $f(x_1)$, $f(x_2)$, and $f(x_u)$. If $f(x_2) < f(x_1)$, then x_l, x_1, x_2 are chosen as new triplets, else if $f(x_2) > f(x_1)$ then x_1, x_2, x_u are chosen. Subsequently, we iterate this process until the outer x-coordinates are closer than ϵ, which is the predetermined minimal distance to terminate the algorithm.

The main advantage of the golden-section search algorithm is the efficiency. The evaluation of $f(x)$ requires a considerable time for the real-world workloads. The golden-section algorithm only requires one additional evaluation for each iteration. Moreover, the complexity of the algorithm is logarithmic. The big-O notation of the algorithm is shown in Equation (1), where N is the number of iterations required to find the solution [26,27].

$$N = O(log\frac{1}{\epsilon}) \tag{1}$$

3.4.1. Considerations for Alternative Algorithms

We choose the golden-section search rather than other line search algorithms, such as the bisection, Newton's, Quasi-Newton's, and Nelder-–Mead methods, because Golden-section searching is unconstrained and derivative-free [28].

First, we need an unconstrained method, because the target function to be searched is not specified. Because NetAP should supports various workloads with different characteristics, we cannot assume any constraints for the searching algorithm. We only assume that the target function is one-dimensional, so we avoid using complex multi-dimensional algorithms, such as the gradient, random search, and Nelder–Mead methods [29].

Second, we need a derivative-free method, because we do not know the complete function in the mathematical form. We only know few points that are collected from the execution results during the round. In addition, the derivative cannot be correctly calculated in the Linux kernel, because the floating point operations are not allowed in the kernel mode in order to maintain the user context in the floating point unit [30]. Thus, we avoid algorithms that are based on derivatives, such as the bisection search, Newton's method, and Quasi-Newton's method [31].

Algorithm 1: Network packet handler with periodical polling

```
Data:
round indicates the number corresponding to the current round,
round_length is a predefined length of a round in seconds,
next_time indicates the next time for updating the interval,
current_time() returns a current wall-clock time,
bytes is the sizes of processed packets in the current round,
interval is a polling interval in the current round,
MaxRound is the maximum number of rounds without changing the interval,
MinChange is the minimum ratio of processed packet sizes between previous and current rounds,
RX_queue, and TX_queue are the queues for the packets received and packets to be transmitted.
1  Function the_packet_handler_of_a_host_system(void):
2      start:
3      next_time ← current_time();
4      round ← 0;
5      bytes ← 0;
6      interval ← 0;
7      while true do
8          Disable interrupts for packet arrivals;
9          if current_time() >= next_time then
10             interval ← golden_section_searching(round, bytes, interval);
11             if interval is not changed AND bytes is changed more than MinChange then
12                 go to start;
13             end
14             next_time ← current_time() + round_length;
15             bytes ← 0;
16             round ← round + 1;
17             if round = MaxRound then
18                 go to start;
19             end
20         end
21         bytes ← bytes + handle_packets_in_the_queue(RX_queue);
22         bytes ← bytes + handle_packets_in_the_queue(TX_queue);
23         Enable interrupts;
24         Sleep for interval;
25     end
26 end
27 Function handle_packets_in_the_queue(queue):
28     processed_bytes = 0;
29     while queue has a packet do
30         Lock the queue;
31         Dequeue a packet;
32         Process the packet;
33         processed_bytes is increased by the size of the packet;
34         Unlock the queue;
35     end
36     return processed_bytes
37 end
38 Function current_time(void):
39     return Current wall-clock time;
40 end
```

3.4.2. Preliminary Examination for Golden-Section Search Algorithm

Before we apply the algorithm to solve our problem, there are two requirements to examine: the target function must be unimodal, and a near-optimal value must exist within a range of possible intervals to perform a search. Thus, we demonstrate in advance whether our problem can be solved by the golden-section search algorithm.

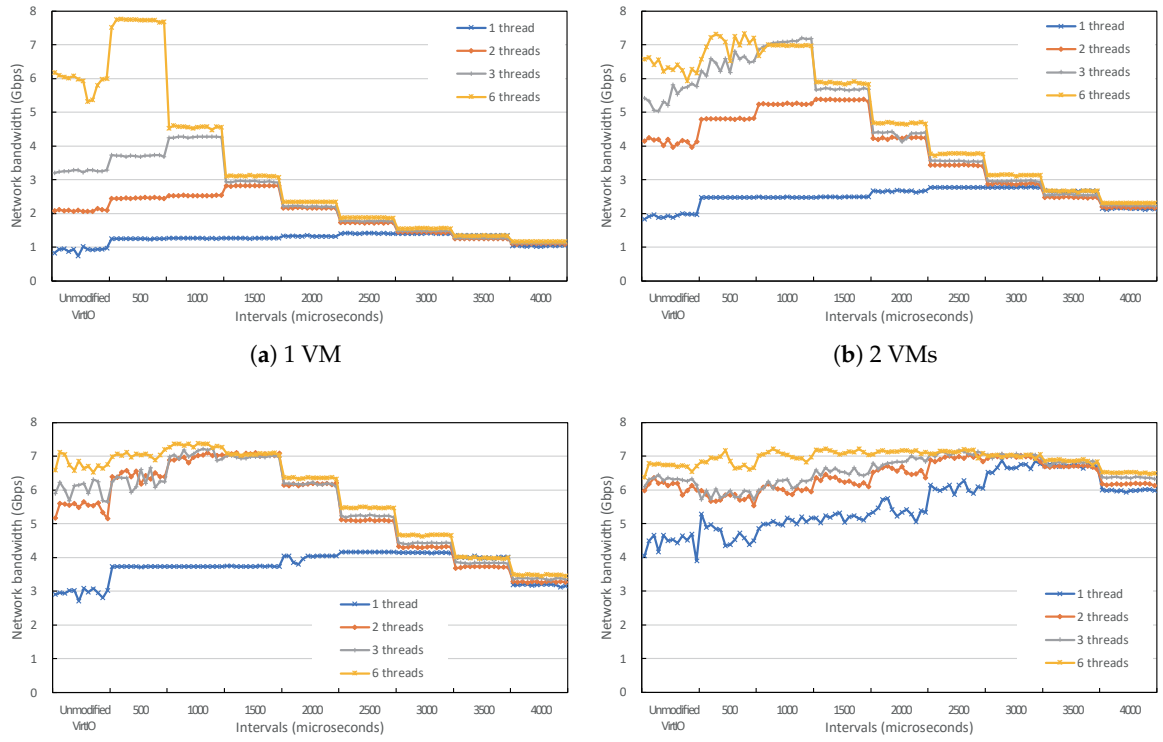
We performed a preliminary examination to determine whether a golden-section search can find near-optimal polling intervals. As a result, we made two observations. First, the change in performance over various polling intervals appears as a unimodal function. Second, there is a range of periods that can be commonly applied to various workloads.

The environment of the experiment is as follows: the host system is equipped with a CPU with six physical cores, 16 GB of memory, and a 10 GbE NIC. Simultaneous multi-threading was disabled, and an additional 1 GbE NIC for controlling the host system was installed to increase the reliability of the experiment. The Linux kernel used corresponded to version v5.1.5, released in May 2019. We ran several VMs on this host system and performed *Netperf* benchmarks on each VM that was connected to a separate Netperf server system. In the experiment, 64B packets were transmitted using TCP, and we measured the aggregated network bandwidth in Mbps.

In this preliminary experiment, We executed one, two, three, and six Netperf threads on each of one, two, three, and six VMs. We increased the polling interval by 500 every 60 s, from 500 us to 5000 us, and observed the performance change that corresponded to each interval for each workload. We did not evaluate each interval separately, but continuously changed the interval. This is to determine how rapidly the effect of changing the polling interval becomes apparent.

Figure 2 shows the result. The first item on the X-axis shows the performance of unmodified VirtIO without applying the periodic polling technique. Furthermore, the figure shows the performance that corresponds to up to 5000 us with gradual increments of 500 us. The performance is shown at five-second intervals. The overall trend forms a unimodal function in every case, and we can find the point where the performance is maximized. As summarized in Table 1, intervals up to a maximum of 3000 us maximized the performance for all workloads. The bandwidth comparison in Table 1 is the relative performance of the best network bandwidth as compared to the performance of unmodified VirtIO.

Through the preliminary examination, we determined that the golden-section search algorithm can be applied to find the near-optimal polling interval in NetAP. We also found that the network performance changed immediately after the polling interval was changed.



(**a**) 1 VM (**b**) 2 VMs

(**c**) 3 VMs (**d**) 6 VMs

Figure 2. Results of preliminary examination for four scenarios.

Table 1. Polling interval maximizing the bandwidth, and comparison with unmodified VirtIO.

	Best Polling Interval (Us)				Bandwidth Comparison			
Threads	**1**	**2**	**4**	**6**	**1**	**2**	**4**	**6**
1 VM	2500	1500	500	500	136%	139%	119%	128%
2 VMs	3000	1500	500	500	142%	131%	116%	107%
3 VMs	3000	2500	2000	1000	163%	142%	111%	109%
6 VMs	3000	2500	2000	1500	163%	142%	111%	109%

3.4.3. Golden-Section Search Algorithm

Algorithm 2 obtains the near-optimal interval using golden-section search. First, R is the so-called golden ratio, which is 0.618, and $MaxInterval$ and $MinInterval$ are the predetermined maximum and minimum intervals, respectively. $MinDistance$ is a predetermined minimum distance to stop the algorithm.

The $golden\_section\_searching()$ function is called by Algorithm 1 with the round, bytes, and previous interval as arguments. In the first and second rounds, $interval_{low}$ and $interval_{high}$ are set using $MaxInterval$ and $MinInterval$, respectively, with the same values of $distance$, and the performance is collected for a $round\_length$. From the third round, the function checks which value of $interval_{low}$ or $interval_{high}$ achieved higher performance. If $interval_{low}$ has higher performance, $interval_{high}$ is replaced by $interval_{low}$, and $interval_{low}$ is replaced by $(interval_{high} + distance)$. Conversely, if $interval_{high}$ provides higher performance, $interval_{low}$ is replaced by $interval_{high}$, and $interval_{high}$ is replaced by $(interval_{low} - distance)$. Subsequently, the updated interval is returned. Furthermore, $distance$ is decreased by the golden ratio in each round. When $distance$ is less than $MinDistance$, the algorithm terminates, and the previous interval is returned without an update.

Through the aforementioned process, we can quickly search for the interval between $MinInterval$ and $MaxInterval$ that maximizes the performance.

Algorithm 2: Golden-section search algorithm to find the near-optimal interval

Data: $interval_{low}$, $interval_{high}$, $perf_{low}$, $perf_{high}$, $temp$, $distance$, $selected$, R, $MinDistance$, $MaxInterval$ and $MinInterval$.
Input: $round, bytes, interval$
Output: The next polling interval

Function $golden\_section\_searching(void)(round,bytes,interval)$**:**
  if $round = 0$ **then**
    $distance \leftarrow R \times (MaxInterval - MinInterval)$;
    $interval_{low} \leftarrow MinInterval + distance$;
    return $interval_{low}$
  end
  if $round = 1$ **then**
    $perf_{low} \leftarrow bytes$;
    $interval_{high} \leftarrow MaxInterval - distance$;
    $selected \leftarrow High$;
    return $interval_{high}$
  end
  if $selected = Low$ **then**
    $perf_{low} \leftarrow bytes$;
  else
    $perf_{high} \leftarrow bytes$;
  end
  if $distance <= MinDistance$ **then**
    return $interval$
  end
  $distance \leftarrow distance \times R$;
  if $perf_{low} > perf_{high}$ **then**
    $temp \leftarrow interval_{high}$;
    $inverval_{high} \leftarrow interval_{low}$;
    $interval_{low} \leftarrow temp + distance$;
    $perf_{high} \leftarrow perf_{low}$;
    $selected \leftarrow Low$;
    return $interval_{low}$
  else
    $temp \leftarrow interval_{low}$;
    $inverval_{low} \leftarrow interval_{high}$;
    $interval_{high} \leftarrow temp$ - $distance$;
    $perf_{low} \leftarrow perf_{high}$;
    $selected \leftarrow High$;
    return $interval_{high}$
  end
end

For the implementation in the Linux kernel, R and $distance$ were calculated and entered in advance as constant values. This is because floating-point operations cannot be performed within the Linux kernel. $MinInterval$ and $MaxInterval$ were set to 100 and 4000, respectively. The numbers are

chosen according to the results of our preliminary examination, which showed that the near-optimal polling intervals can be found between 500 and 3000 us. In addition, *MinDistance* was set to 30, so that the number of searches was limited to 11 iterations. The set of all distances is as follows: 2410, 1490, 921, 569, 352, 217, 134, 83, 51, 32. Note that the first distance is used twice, at the first and second rounds.

4. Evaluation

We implemented NetAP on Linux v5.1.5 and evaluated the performance. We attempted to answer the following questions.

- What is the most proper round length for performance collection and golden-section search algorithm? (Section 4.1)
- How does the polling interval change at runtime? (Section 4.2)
- What is the impact of NetAP on CPU utilization and response time? (Section 4.2)
- Is NetAP also effective for various packet sizes and different system environments? (Section 4.3)

The experimental environment was the same as that described in Section 3.4.2. In addition, for all experiments, the existing unmodified VirtIO processing was run at the beginning. Subsequently, we applied NetAP after 30 s (in Section 4.1) or 20 s (in Section 4.2 and 4.3) to observe the impact of NetAP on the performance of VMs easily. In the first and second experiments, we measured network bandwidth for 64B packets only. This was because a substantial amount of data center traffic consists of small packets, which consume more CPU resources than large packets [32,33]. Furthermore, *perf* and *pidstat* were used to measure CPU utilization.

4.1. First Experiment: Round Length

The first experiment evaluated the adaptation performance of NetAP depending on the round length. To find the near-optimal polling interval, the golden-section searching algorithm collects performance over a period (i.e., a round length). Subsequently, the changed interval is applied to VirtIO processing until the next round. If the round is very short, the polling interval changes frequently, which makes it difficult for the algorithm to find the near-optimal polling interval. In contrast, a long round length can degrade the average performance because inappropriate polling intervals can affect the performance negatively. As a result, it is necessary to determine a proper round length for the golden-section search algorithm.

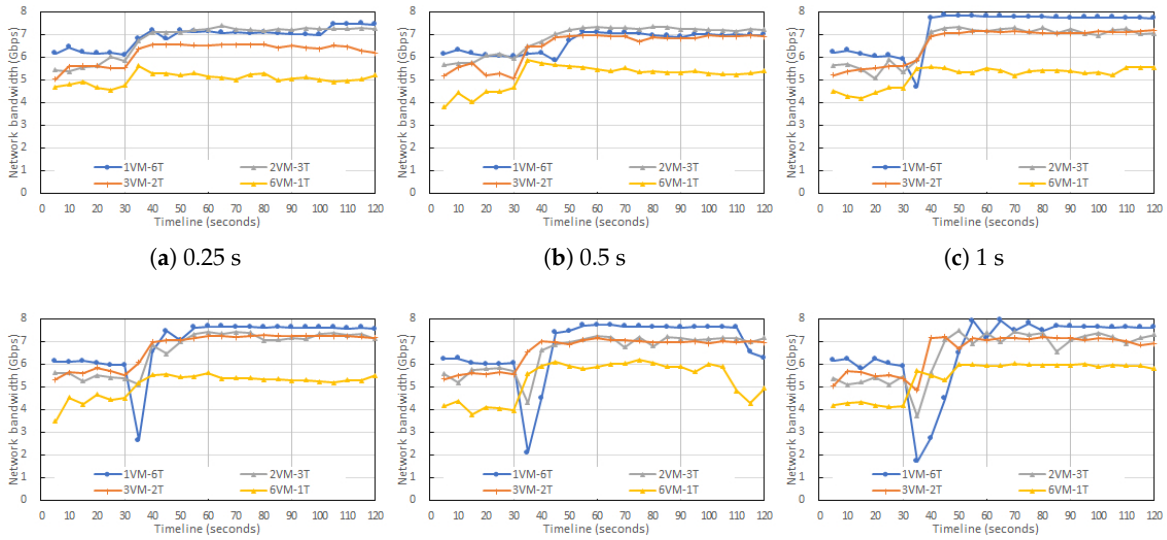
We have four experimental scenarios: 1VM-6T, 2VM-3T, 3VM-2T, and 6VM-1T. This indicates the number of Netperf threads and the number of VMs on which the aforementioned threads are executed. For example, 3VM-2T means that two Netperf threads run on each of three VMs. In total, six Netperf threads and three vHost threads are executed simultaneously for the 3VM-2T scenario. We ran all experiments for 120 s, and unmodified VirtIO processing was executed in the first 30 s. Subsequently, we measured the bandwidth of the 10G NIC shared by all VMs every 5 s. The lengths of the rounds selected for the experiment were 0.25, 0.5, 1, 2, 3, and 5 s.

Figure 3 shows the performance change in aggregate bandwidth, depending on round length. In most cases, NetAP shows performance improvement compared to the unmodified VirtIO running for the first 30 s of the experiment. In addition, we observe that the performance of VMs decreases during the time for the searching algorithm to find the near-optimal polling interval when the round length is longer than one second. This is the aforementioned negative effect of a long round length. During the long round, an inappropriate value calculated by the golden-section search algorithm may be applied to the polling interval; this will result in performance degradation.

Table 2 depicts the performance shown in Figure 3 relative to the best bandwidth measured in Table 1. We divide the results presented in Figure 3 into two parts: the second quarter (30–60 s) and the last quarter (90–120 s). The last quarter shows the performance when the adaptation is completed, and the second quarter shows the performance when the negative effect of adaptation is maximized.

In Table 2, we find that NetAP can find the near-optimal polling interval and achieve a bandwidth close to that of the best case when the round length is longer than one second. However, in 6VM-1T, the relative performance is less than 90% for all round lengths, which indicates that NetAP has difficulty in finding the near-optimal polling interval. We assume the reason is that the golden-section search algorithm is executed in a distributed manner; furthermore, it does not consider the effects of the interference between multiple vHost threads that are executed in parallel. We plan to overcome this limitation of NetAP by enhancing the algorithm in future work.

In addition, Table 2 shows the performance under different round lengths during adaptation (i.e., 30–60 s). Unlike the results after adaptation (i.e., 90–120 s), the performance obtained for round lengths longer than two seconds is relatively low. This is because the inappropriate polling interval reduces the performance of VMs for a long period, as illustrated in Figure 3. Based on this result, we selected one second as the default round length for NetAP and performed the following experiments.



(**a**) 0.25 s (**b**) 0.5 s (**c**) 1 s

(**d**) 2 s (**e**) 3 s (**f**) 5 s

Figure 3. Aggregate bandwidth of VMs with NetAP under the different round lengths.

Table 2. Performance relative to the best bandwidth in Figure 2.

	(a) After Adapting (90 s–120 s)					**(b) While Adapting (30 s–60 s)**			
	1VM-6T	**2VM-3T**	**3VM-2T**	**6VM-1T**		**1VM-6T**	**2VM-3T**	**3VM-2T**	**6VM-1T**
0.25 s	94.54%	102.53%	90.23%	74.96%	0.25 s	91.15%	99.94%	92.24%	78.91%
0.5 s	90.56%	101.69%	97.91%	78.96%	0.5 s	84.44%	98.95%	95.97%	84.00%
1 s	100.27%	100.15%	100.89%	80.58%	1 s	94.32%	98.76%	97.13%	81.30%
2 s	98.35%	102.49%	102.19%	78.84%	2 s	84.16%	94.38%	97.98%	81.35%
3 s	93.55%	100.39%	98.81%	78.44%	3 s	79.58%	92.02%	98.24%	87.28%
5 s	98.80%	101.59%	99.28%	88.13%	5 s	65.79%	89.90%	94.62%	85.35%

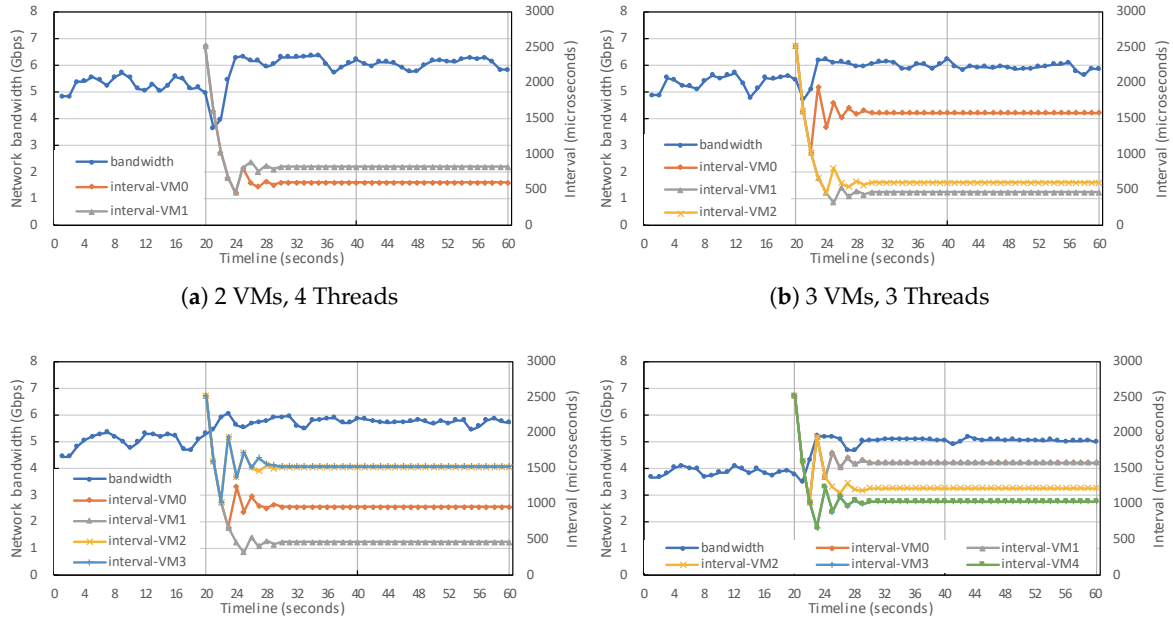
4.2. Second Experiment: CPU Utilization and End-User Latency

This subsection identifies the impact of NetAP on CPU utilization and end-user latency. We constructed four scenarios, 2VM-4T, 3VM-3T, 4VM-2T, and 5VM-1T, which were not tested in Section 4.1. We measured the aggregate bandwidth and polling intervals every second for 60 s. In this experiment, NetAP starts to run at 20 s after the unmodified VirtIO has run for the first 20 s.

Figure 4 illustrates the aggregate network bandwidth (left y-axis) and the polling interval (right y-axis) of each VM over time. In all scenarios, the aggregate bandwidth increases when NetAP is applied at 20 s. NetAP periodically checks whether there are pending packets for processing before a vIRQ is generated. This procedure prevents expensive VM-Exit and VM-Entry operations and improves the overall network performance. Furthermore, we find that NetAP offers more stable performance than the unmodified VirtIO. In the unmodified VirtIO, a vIRQ to notify the vHost thread can be generated after the corresponding VM is scheduled by the CPU scheduler. Therefore, the vIRQ can be delayed, and this may cause unstable performance.

In the figure, the polling interval of each VM varies for approximately 10 s when NetAP is applied. This is because the algorithm searches for the near-optimal polling 11 times for an adaptation. After the near-optimal interval is found, NetAP continuously maintains the interval.

In addition, the near-optimal polling interval of each VM varies in Figure 4 when multiple VMs run concurrently. As described in Section 4.1, the reason is that the golden-section searching algorithm is executed in a distributed-manner. Each searching algorithm separately runs per vHost thread during the round. However, NetAP still improves the aggregate bandwidth of VMs by adjusting the polling intervals of each VM independently.



(**a**) 2 VMs, 4 Threads (**b**) 3 VMs, 3 Threads

(**c**) 4 VMs, 2 Threads (**d**) 5 VMs, 1 Thread

Figure 4. Aggregate bandwidth and the polling interval of each VM over time in different experiment scenarios.

Tables 3 and 4 show the aggregate bandwidth, CPU utilization, and latency for every 20 s in each scenario; 0 to 20 s is the time for which unmodified VirtIO runs (VirtIO period), 20 to 40 s is when the algorithm searches for the near-optimal polling interval (Adapting period), and the last 20 s correspond to the time after the near-optimal interval has been applied (Adapted period). First, NetAP in the Adapted period shows that the aggregate bandwidth is improved by up to 30.37% compared to that in the VirtIO period. In each Adapted period, the CPU utilization of the host is decreased compared to that in the VirtIO period because NetAP reduces the number of VM-Exit and VM-Entry operations that occur in the case of the interrupt-based method in the existing VirtIO. In addition, NetAP increases the guest CPU utilization as NetAP processes incoming packets on a timely basis and allows the corresponding VM to generate more packets in the queue. As a result, NetAP achieves improved efficiency in packet processing.

In addition, end-user latency decreases by up to 89.49%. The standard deviation in the Adapted period also reduces to a maximum level of 40%, and it shows improvement compared to the VirtIO period for all scenarios. From these results, we can assume that the time for delivering vIRQs is variable in the unmodified VirtIO. When the corresponding VM is selected by the CPU scheduler, the VM can generate a vIRQ, Therefore, the time is not deterministic. This causes fluctuating end-user latency. In contrast, NetAP offers stable and low latency by operating at periodic intervals determined by the searching algorithm, thereby increasing the efficiency of packet processing.

Table 3. Detailed performance and CPU utilization in for 2VM-4T and 3VM-3T.

	2 VMs, 4 Threads Each				3 VMs, 3 Threads Each			
Period	**VirtIO**	**Adapting**	**Adapted**	**Cf.**	**VirtIO**	**Adapting**	**Adapted**	**Cf.**
Bandwidth (Gbps)	5.27	5.88	6.05	114.91%	5.34	5.93	5.89	110.31%
CPU-guest (%)	549.36	552.66	570.06	103.77%	561.66	568.44	574.86	102.35%
CPU-host (%)	36.36	11.34	12.78	35.15%	31.32	18.24	17.10	54.60%
Latency-mean (us)	0.76	0.67	0.67	87.21%	0.85	0.76	0.76	89.49%
Latency-stddev (us)	49.69	53.87	47.92	96.44%	60.86	61.35	54.06	88.82%

Table 4. Detailed performance and CPU utilization in 4VM-1T and 5VM-1T.

	4 VMs, 2 Threads Each				5 VMs, 1 Thread Each			
Period	**VirtIO**	**Adapting**	**Adapted**	**Cf.**	**VirtIO**	**Adapting**	**Adapted**	**Cf.**
Bandwidth (Gbps)	5.02	5.76	5.73	114.16%	3.86	4.93	5.04	130.37%
CPU-guest (%)	558.42	572.88	573.24	102.65%	513.06	520.50	519.30	101.22%
CPU-host (%)	32.10	19.62	21.36	66.54%	72.00	37.98	39.54	54.70%
Latency-mean (us)	0.79	0.69	0.69	87.36%	0.65	0.50	0.49	75.93%
Latency-stddev (us)	52.08	45.73	44.24	84.94%	24.69	12.31	10.08	40.82%

4.3. Third Experiment: Different Packet Sizes and System Environments

In this experiment, we evaluated NetAP for different packet sizes and system environments. We intended to demonstrate that NetAP with the current parameters is effective in different environments, and that, consequently, many of the pre-examination and workload analysis steps are not mandatory.

First, we evaluated NetAP for various packet sizes: 128, 256, 512, and 1024 B. In this experiment, we ran a single VM executing Netperf while using a single thread for 60 s and applied NetAP after 20 s, the same as in the previous experiments described in Section 4.2. Figure 5 shows that performance decreases briefly when NetAP starts to run at 20 s in order to determine the near-optimal interval; then, the performance increases significantly. Table 5 depicts the relative bandwidth, CPU utilization, and latency compared to the unmodified VirtIO, which shows that NetAP improves the performance by at least 53.53% and by a maximum of 148.83%. The results of unmodified VirtIO were collected in the first 20 s, and the results of NetAP were collected in last 20 s. The CPU utilization of the host drastically decreased while the overall performance increased. Such efficient CPU utilization is an important factor in the cloud system, which indicates that NetAP more efficiently handles network packets, allowing for other CPU-intensive threads or different VMs to use more CPU resources. In addition, the average latency decreased by 15.36%. For 256-B packets, the standard deviation of latency increased by 15.36%, but the average latency decreased significantly, by 52.22%. As a result, NetAP provides higher network performance with fewer CPU resources for both large packets and the small packets that incur severe CPU contention.

Second, we constructed an additional experimental setup, as follows: two servers equipped with Intel Xeon E5-2650 v2 CPUs (8 cores, 3.4 GHz) and 64 GB of memory were connected to a 10 GbE

switch. The VM specifications and the network configurations are the same as in previous experiments. This experimental setup shows the impact of CPU performance, which has a critical effect on network performance in a virtualized environment. In terms of BogoMips, a single CPU core that was used in the previous experiment recorded 6192, whereas a single CPU core in this experiment recorded 5200. Thus, the CPUs that wew used in this experiment approximately exhibit a 20% lower performance. We applied NetAP to this system without additional modifications. The experimental configuration and method are identical to those presented in Section 4.2.

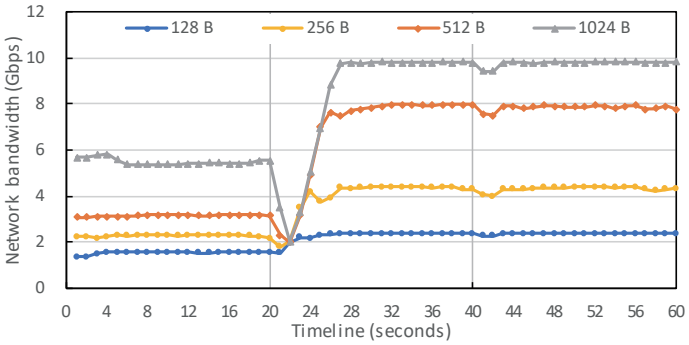


Figure 5. Aggregate bandwidth over time for different packet sizes.

Table 5. Detailed performance and CPU utilization for different packet sizes.

	128 B	**256 B**	**512 B**	**1024 B**
Bandwidth	153.53%	190.05%	248.83%	177.88%
CPU-guest	52.03%	54.09%	61.54%	47.96%
CPU-host	8.39%	16.47%	22.01%	15.77%
Latency-mean	64.62%	52.22%	61.18%	100.00%
Latency-stddev	41.00%	115.36%	12.69%	99.03%

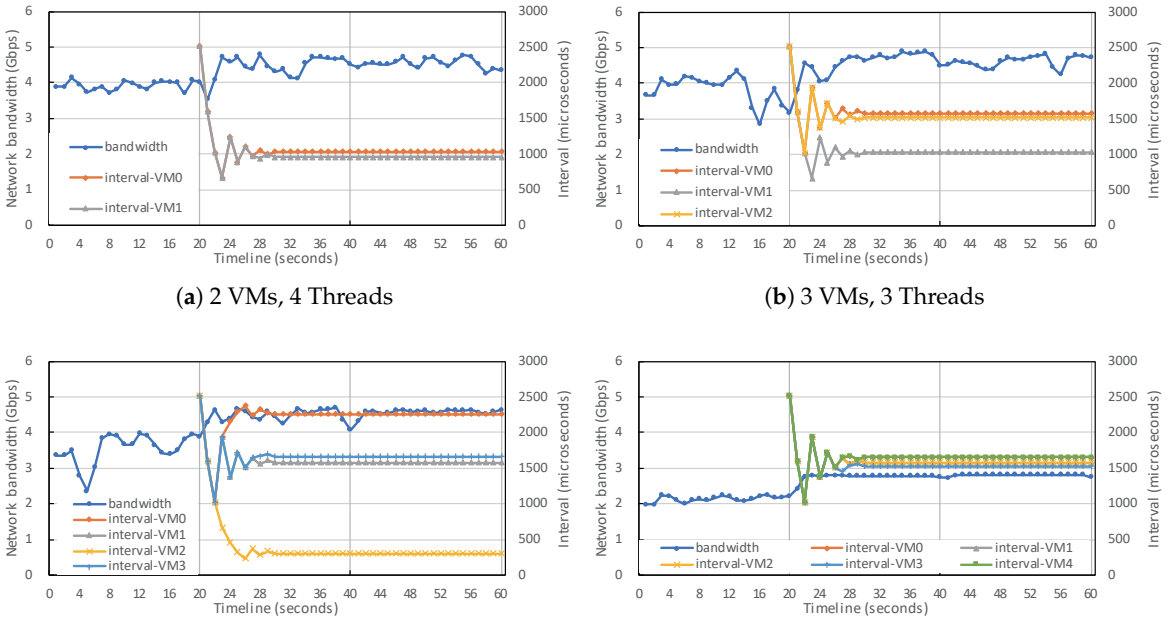
Figure 6 shows similar results to Figure 4, which indicates that NetAP improved the aggregate bandwidth as compared to the unmodified VirtIO. However, the overall network performance was lower than that shown in Figure 4. This is because the performance of the CPU cores used in this experiment is lower than that of the ones used to obtain the results that are shown in Figure 4. Tables 6 and 7 show that the performance of NetAP improved by up to 31.16%, while showing results that are similar to those in Tables 3 and 4 for CPU utilization and latency. These results show that NetAP is also effective in systems with different hardware specifications. Thus, we can conclude that the NetAP polling mechanism with the default parameters is effective in various system environments.

Table 6. Detailed performance and CPU utilization in 2VM-4T and 3VM-3T for the additional experiment setup.

	2 VMs, 4 Threads Each				**3 VMs, 3 Threads Each**			
Period (s)	**0–20**	**20–40**	**40–60**	**Comparison**	**0–20**	**20–40**	**40–60**	**Comparison**
Bandwidth (Gbps)	3.92	4.46	4.54	115.78%	3.81	4.59	4.61	120.84%
CPU-guest (%)	738.16	747.36	764.00	103.50%	731.44	768.72	765.52	104.66%
CPU-host (%)	49.36	20.40	20.40	41.33%	55.68	15.44	18.40	33.05%
Latency-mean (us)	1.00	0.88	0.87	86.72%	1.17	0.95	0.96	82.13%
Latency-stddev (us)	30.74	33.10	20.09	65.36%	51.06	45.99	44.34	86.84%

Table 7. Detailed performance and CPU utilization in 4VM-1T and 5VM-1T for the additional experiment setup.

	4 VMs, 2 Threads Each				5 VMs, 1 Thread Each			
Period (s)	**0–20**	**20–40**	**40–60**	**Comparison**	**0–20**	**20–40**	**40–60**	**Comparison**
Bandwidth (Gbps)	3.54	4.48	4.58	129.15%	2.14	2.76	2.81	131.16%
CPU-guest (%)	708.88	761.52	762.00	107.49%	617.28	552.32	603.76	97.81%
CPU-host (%)	74.40	26.32	25.60	34.41%	102.56	34.88	33.76	32.92%
Latency-mean (us)	0.98	0.83	0.85	87.07%	1.19	0.91	0.92	76.88%
Latency-stddev (us)	40.42	19.74	20.29	50.18%	6.97	2.58	2.08	29.77%



(**a**) 2 VMs, 4 Threads

(**b**) 3 VMs, 3 Threads

(**c**) 4 VMs, 2 Threads

(**d**) 5 VMs, 1 Thread

Figure 6. Aggregate bandwidth and polling interval of each VM over time for the additional environment.

5. Conclusions

This paper presented NetAP, an adaptive polling technique for improving the performance of packet processing in VMs. The periodic polling approach of NetAP resolves the inefficient packet processing of the existing VirtIO, which is interrupt-based. We used the golden-section search algorithm and implemented NetAP on KVM hypervisor to find the optimal polling interval for NetAP. We implemented NetAP in Linux v5.1.5 and evaluated it with up to six VMs. The evaluation results showed that NetAP improves the network performance of VMs by up to 31.16%, while using the host CPU for packet processing at only 32.92% of the usage exhibited by the existing VirtIO technique.

In this paper, we discovered areas for future work in adaptive polling-based network packet processing. First, accurate control or adaptation for the performance isolation between VMs should be provided for real-world cloud systems. When the VMs share the same host machine, the workloads of the VMs influence each other. This "neighborhood effect" of the virtualized system incurs the performance interference between VMs on the same host machine. When the performance of a VM decreases due to other co-located VMs, the golden-section search algorithm receives erroneous performance by the neighborhood effect and it has difficulty in finding the near-optimal polling interval to achieve the maximum performance.

Another issue is the estimation of the optimal polling interval. If we can determine the optimal polling interval for a workload without carrying out adaptation, the performance degradation during

adaptation will disappear. The data that can be employed for the estimation of the optimal polling interval include the average packet size, packet incoming rate, current throughput of the physical NIC, and the number of vHost and client threads. The correct model cannot be easily developed because several of the measurements are related to the interval. In our future work, we aim to develop an AI-based model in order to solve this problem.

Author Contributions: The work presented here was completed in collaboration between all authors. conceptualization, H.P., K.L., and C.-H.H.; validation, H.P., K.L., and C.-H.H.; formal analysis, H.P.; investigation, H.P., J.S. and M.L.; writing–original draft preparation, H.P., K.L., and C.-H.H.; writing–review and editing, H.P., K.L., J.S., M.L., and C.-H.H.; supervision, H.P. and C.-H.H., funding acquisition, H.P. and C.-H.H. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by a National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIP) (No. NRF-2020R1F1A1067365). This research was also supported by the Chung-Ang University Research Scholarship Grants in 2019.

Acknowledgments: The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of this paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Menon, A.; Santos, J.R.; Turner, Y.; Janakiraman, G.; Zwaenepoel, W. Diagnosing performance overheads in the xen virtual machine environment. In Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments, Chicago, IL, USA, 11–12 June 2005; pp. 13–23.
2. Liu, J.; Huang, W.; Abali, B.; Panda, D.K. High Performance VMM-Bypass I/O in Virtual Machines. In Proceedings of the USENIX Annual Technical Conference, General Track, Boston, MA, USA, 30 May–3 June 2006; pp. 29–42.
3. Russell, R. virtio: Towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Oper. Syst. Rev.* **2008**, *42*, 95–103. [CrossRef]
4. Motika, G.; Weiss, S. Virtio network paravirtualization driver: Implementation and performance of a de-facto standard. *Comput. Stand. Interfaces* **2012**, *34*, 36–47. [CrossRef]
5. Adams, K.; Agesen, O. A comparison of software and hardware techniques for x86 virtualization. *ACM Sigplan Not.* **2006**, *41*, 2–13. [CrossRef]
6. Ben-Yehuda, M.; Day, M.D.; Dubitzky, Z.; Factor, M.; Har'El, N.; Gordon, A.; Liguori, A.; Wasserman, O.; Yassour, B.A. The Turtles Project: Design and Implementation of Nested Virtualization. *Osdi* **2010**, *10*, 423–436.
7. Lettieri, G.; Maffione, V.; Rizzo, L. A study of I/O performance of virtual machines. *Comput. J.* **2018**, *61*, 808–831. [CrossRef]
8. Yang, J.; Minturn, D.B.; Hady, F. When poll is better than interrupt. *FAST* **2012**, *12*, 3.
9. Kohler, E.; Morris, R.; Chen, B.; Jannotti, J.; Kaashoek, M.F. The Click modular router. *ACM Trans. Comput. Syst. (TOCS)* **2000**, *18*, 263–297. [CrossRef]
10. Intel, D. *Data Plane Development Kit*; 2014. Available online: https://www.dpdk.org/ (accessed on 29 July 2020).
11. Kim, T.Y.; Kang, D.H.; Lee, D.; Eom, Y.I. Improving performance by bridging the semantic gap between multi-queue SSD and I/O virtualization framework. In Proceedings of the 2015 31st Symposium on Mass Storage Systems and Technologies (MSST), Santa Clara, CA, USA, 30 May–5 June 2015; pp. 1–11.
12. Hajnoczi, S. Applying polling techniques to QEMU. In Proceedings of the KVM Forum 2017, Prague, Czech Republic, 25–27 October 2017; pp. 25–27.
13. Dong, Y.; Yang, X.; Li, J.; Liao, G.; Tian, K.; Guan, H. High performance network virtualization with SR-IOV. *J. Parallel Distrib. Comput.* **2012**, *72*, 1471–1480. [CrossRef]
14. Dong, Y.; Yu, Z.; Rose, G. SR-IOV Networking in Xen: Architecture, Design and Implementation. In Proceedings of the Workshop on I/O Virtualization, San Diego, CA, USA, 10–11 December 2008; Volume 2.
15. Liu, J. Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support. In Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), Atlanta, GA, USA, 19–23 April 2010; pp. 1–12.

16. Gordon, A.; Amit, N.; Har'El, N.; Ben-Yehuda, M.; Landau, A.; Schuster, A.; Tsafrir, D. ELI: Bare-metal performance for I/O virtualization. *ACM SIGPLAN Not.* **2012**, *47*, 411–422. [CrossRef]
17. Abramson, D.; Jackson, J.; Muthrasanallur, S.; Neiger, G.; Regnier, G.; Sankaran, R.; Schoinas, I.; Uhlig, R.; Vembu, B.; Wiegert, J. Intel Virtualization Technology for Directed I/O. *Intel Technol. J.* **2006**, *10*, 179–192. [CrossRef]
18. Landau, A.; Ben-Yehuda, M.; Gordon, A. SplitX: Split Guest/Hypervisor Execution on Multi-Core. In Proceedings of the WIOV, Portland, OR, USA, 14 June 2011.
19. Xu, C.; Gamage, S.; Lu, H.; Kompella, R.; Xu, D. vturbo: Accelerating virtual machine i/o processing using designated turbo-sliced core. In Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13), San Jose, CA, USA, 26–28 June 2013; pp. 243–254.
20. Yasukata, K.; Huici, F.; Maffione, V.; Lettieri, G.; Honda, M. HyperNF: Building a high performance, high utilization and fair NFV platform. In Proceedings of the 2017 Symposium on Cloud Computing, Santa Clara, CA, USA, 24–27 September 2017; pp. 157–169.
21. Hwang, J.; Ramakrishnan, K.K.; Wood, T. NetVM: High performance and flexible networking using virtualization on commodity platforms. *IEEE Trans. Netw. Serv. Manag.* **2015**, *12*, 34–47. [CrossRef]
22. Dovrolis, C.; Thayer, B.; Ramanathan, P. HIP: Hybrid interrupt-polling for the network interface. *ACM SIGOPS Oper. Syst. Rev.* **2001**, *35*, 50–60. [CrossRef]
23. Salah, K.; El-Badawi, K.; Haidari, F. Performance analysis and comparison of interrupt-handling schemes in gigabit networks. *Comput. Commun.* **2007**, *30*, 3425–3441. [CrossRef]
24. Salah, K.; Qahtan, A. Implementation and experimental performance evaluation of a hybrid interrupt-handling scheme. *Comput. Commun.* **2009**, *32*, 179–188. [CrossRef]
25. Kiefer, J. Sequential minimax search for a maximum. *Proc. Am. Math. Soc.* **1953**, *4*, 502–506. [CrossRef]
26. Chong, E.K.; Zak, S.H. *An Introduction to Optimization*; John Wiley & Sons: Hoboken, NJ, USA, 2004.
27. Luenberger, D.G.; Ye, Y. *Linear and Nonlinear Programming*; Springer: Reading, MA, USA, 1984; Volume 2.
28. Nocedal, J.; Wright, S. *Numerical Optimization*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2006.
29. Nelder, J.A.; Mead, R. A simplex method for function minimization. *Comput. J.* **1965**, *7*, 308–313. [CrossRef]
30. The Linux Kernel Documentation. Available online: https://www.kernel.org/doc/html/latest/ (accessed on 29 July 2020).
31. Dennis, J.E., Jr.; Moré, J.J. Quasi-Newton methods, motivation and theory. *SIAM Rev.* **1977**, *19*, 46–89. [CrossRef]
32. Roy, A.; Zeng, H.; Bagga, J.; Porter, G.; Snoeren, A.C. Inside the social network's (datacenter) network. In Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, London, UK, 17–21 August 2015; pp. 123–137.
33. Đukić, V.; Jyothi, S.A.; Karlaš, B.; Owaida, M.; Zhang, C.; Singla, A. Is advance knowledge of flow sizes a plausible assumption? In Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), Boston, MA, USA, 26–28 February 2019; pp. 565–580.