Enhancing Fidelity of P4-Based Network Emulation with a Lightweight Virtual Time System

Gong Chen
gchen31@hawk.iit.edu
Illinois Institute of Technology
Chicago, IL, USA

Zheng Hu
zhenghu@uark.edu
University of Arkansas
Fayetteville, AR, USA

Dong Jin
dongjin@uark.edu
University of Arkansas
Fayetteville, AR, USA

ABSTRACT

P4's data-plane programmability allows for highly customizable and programmable packet processing, enabling rapid innovation in network applications, such as virtualization, security, load balancing, and traffic engineering. Researchers extensively use Mininet, a popular network emulator, integrated with BMv2, for fast and flexible prototyping of these P4-based applications, but due to its lower performance in terms of throughput and latency compared to a production-grade software switch like Open vSwitch, it is crucial to have an accurate and scalable emulation testbed. In this paper, we develop a lightweight virtual time system and integrate it into Mininet with BMv2 to enhance fidelity and scalability. By scaling the time of interactions between containers and the underlying physical machine by a time dilation factor (TDF), we can trade time with system resources, making the emulated P4 network appear to be faster from the viewpoint of the switch/host processes in the container. Our experimental results show that the testbed can accurately emulate much larger networks with high loads, scaled by a factor of TDF with extremely low system overhead.

CCS CONCEPTS

• **Computing methodologies** → **Modeling and simulation**; • **Networks** → *Network performance evaluation*; • **Computer systems organization** → Parallel architectures.

KEYWORDS

Virtual Time, Network Emulation, Linux Container, TimeKeeper, Virtualization, Programming Protocol-independent Packet Processors

ACM Reference Format:
Gong Chen, Zheng Hu, and Dong Jin. 2023. Enhancing Fidelity of P4-Based Network Emulation with a Lightweight Virtual Time System. In *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '23), June 21–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3573900.3591120

1 INTRODUCTION

Modern hyperscale data center, service provider, and carrier networks are increasingly built on open switch hardware and open software running on servers that support virtual network functions. As the demand for higher network throughput and efficiency increases, the traditional fixed-function Application-Specific Integrated Circuit (ASIC) switch is being replaced by the more versatile programmable switch that enables greater control of the data plane [6]. P4 stands for Programming Protocol-independent Packet Processors [14] and it provides a high-level programming language for defining the packet processing behavior in programmable switch ASICs. P4 allows network operators and developers to define how packets are processed in the data plane, including how they are parsed, classified, and forwarded, and therefore, enables the creation of custom network functions, such as firewalls, traffic analyzers, and load balancers, that can be tailored to the specific needs of modern networks.

The rapid advancement of programmable networks largely depends on successfully translating early-stage research concepts into practical applications. A realistic and flexible testing environment is often an indispensable tool in achieving such a transformation. Mininet-BMv2 [9] is a network emulator that supports programmability on both the data plane and control plane and has been widely embraced by the P4 community. It provides a flexible and cost-effective experimental platform to create, test, and evaluate P4 applications and protocols. With lightweight OS-level virtualization technology, Mininet displays good scalability (up to 4096 hosts on a commodity laptop) and functional fidelity by running unmodified code of network applications over the actual Linux kernel. However, BMv2 is not designed to be a production-grade software switch [21]. It is intended to be utilized as a tool for designing, testing, and debugging P4 software. Consequently, BMv2's performance, such as throughput and latency, is considerably lower than that of a production-grade software switch such as Open vSwitch (OVS) [25]. This limitation can significantly impact the temporal fidelity of Mininet, particularly when emulating high-workload network scenarios.

To enhance temporal fidelity in network emulation experiments, researchers have developed virtual time systems for virtual machines (VMs) [26] and containers [20]. These systems allow each virtual node to have an independent clock that can advance at a customized rate different from the wall clock. As a result, the nodes can perceive their own notion of time as if they are running concurrently on multiple physical machines, rather than relying on the system clock. In this paper, we design and integrate a virtual time system into the Mininet-BMv2 testbed to improve accuracy in time-dependent P4-based network applications and scenarios.

We first present empirical observations of the temporal fidelity issues in the existing Mininet-BMv2. We then conduct an error analysis of Mininet-BMv2 based on these observations. In order to tackle this problem, we have created and integrated a virtual time system into Mininet-BMv2. The virtual time system assigns a virtual clock to each node within Mininet-BMv2, which operates independently of the physical platform's system time. The system is controlled by a barrier-based synchronization controller that ensures that all nodes are synchronized in virtual time, which helps maintain high-performance fidelity and scalability. Throughout the paper, we refer to the resulting virtual-time-enhanced P4 network emulator as VT-BMv2. We evaluate the performance fidelity, synchronization overhead, and system scalability of VT-BMv2 and compare the measurements to those obtained from Mininet-BMv2. Our results show that VT-BMv2 significantly reduces the error of throughput measurement to a certain percentage, while Mininet-BMv2 has an error of up to another 82.8% in a 16-switches network configuration. VT-BMv2 is a lightweight modification of the Linux Kernel as the experimental results show limited synchronization overhead. For instance, the synchronization overhead of 256 containers on 16 CPUs is less than 0.31 ms per cycle, which is roughly 0.13% of the overall execution time. Meanwhile, VT-BMv2 maintains a stable and high precision of time even with a large number of containers. For example, VT-BMv2 can maintain an average TCP throughput of approximately 97.54% of the desired rate (1000 Mpbs) during the emulation of a 256-switch network with a high volume of TCP transmissions.

The remainder of the paper is structured as follows: Section 2 discusses the background of P4 and the related emulation testbeds. Section 3 presents a model and analysis of the temporal error of Mininet-BMv2. Section 4 outlines the design architecture of VT-BMv2, a virtual time system that incorporates precise time management for Mininet-BMv2 as well as implementation details of VT-BMv2, such as the synchronization controller. Section 5 provides the performance evaluation of VT-BMv2, including performance fidelity, synchronization overhead, and scalability. Section 6 discusses the related work on virtual time systems and container-based network emulators. Section 7 concludes the paper with a discussion of future work.

2 BACKGROUND

2.1 Programming Protocol-independent Packet Processors (P4)

A P4 switch[14] is a specialized type of network switch that leverages the P4 language to define and customize the forwarding behavior of network packets. P4 is a domain-specific language designed specifically for programming data-plane processing functions in networking devices such as switches, routers, and network interface cards [14].

P4 switches exhibit several unique advantages. First, they provide a high degree of programmability and flexibility. Network engineers can specify how packets should be processed and forwarded in the network, including how to classify packets, modify packet headers, and direct packets to specific ports or network functions. P4 switches can be used to implement network functions such as firewalls, load balancers, and intrusion detection systems. Second, P4 hardware switches provide high-performance packet processing and forwarding capability, making them suitable for use in high-speed networking applications such as data centers and 5G telecommunications networks. [24]

P4 has gained widespread attention in recent years as the demand for programmable network devices has increased. Some examples of P4 hardware switches include the Barefoot Tofino switch ASIC[5], the Intel Tofino 2 switch ASIC[8], and the Stratum switch system[10] from the Open Networking Foundation. These switches are specifically designed to be used in large-scale data centers and cloud computing environments, where high-performance and programmable network devices are essential to meet the demands of modern applications and services. For example, the Aurora 710[4] is a P4 hardware switch based on Barefoot Tofino[5] switching silicon. It has 32 QSFP28 interfaces, each capable of supporting 100 Gbps line rate communication.

2.2 Emulation Testbed for Programmable Networks

As the adoption of P4-based network applications grows, it becomes crucial to validate and test them in a realistic environment before deploying them in production. This is because P4 allows for the creation of highly customized behaviors, which can have a significant but unknown impact on network performance and reliability. Hardware testbeds are commonly used, but they are expensive and limited in size. Virtualized networking environments, like Mininet-BMv2 [9], are often used for the initial validation of P4 applications, allowing for the creation of a realistic network environment with multiple P4 switches, hosts, and even programmable controllers.

Mininet is a popular network emulator that utilizes Linux containers to run scalable network experiments. With Mininet, users can create a virtual network environment for testing, development, and evaluation. Mininet supports OpenFlow-enabled switches [23] that operate in kernel space for efficiency, such as Open vSwitch [25]. These switches are virtually linked and connected to containers through virtual interfaces, creating an emulated network topology. This makes Mininet a valuable tool to emulate complex network environments and study the behavior of networks and protocols in a controlled environment. While Mininet does not natively support P4 switches, tools like BMv2 [7] enable P4 switches to be used with Mininet through the P4 Runtime environment and P4 compiler. This makes Mininet a valuable tool for studying the behavior of P4-based networks in a controlled environment.

BMv2 is an open-source software switch built on top of the P4 programming language. It is a flexible platform for designing and testing custom packet-processing pipelines and specifying how packets should be processed and forwarded. BMv2's simplicity and programmability make it a popular tool for researchers and network developers to experiment with new network architectures and protocols without the need for expensive hardware.

Mininet-BMv2 provides high functional fidelity to P4 applications by accurately implementing the P4 language specifications. However, BMv2's performance in terms of throughput and latency is generally slower compared to hardware switches or production-grade software switches like Open vSwitch [25]. This performance gap may create temporal fidelity issues in network emulation when using Mininet-BMv2.

Mininet-BMv2 also faces scalability challenges due to the imprecise time management and process scheduling of Linux container technology. Mininet employs Linux container technology, an OS-level virtualization technique that enables a group of processes, including virtual hosts and application processes running inside the container, to have an independent view of system resources, such as process ID, file system, and network interfaces, and shares the kernel with other containers. In contrast to VM-based virtualization techniques where each virtual node has a copy of the entire OS kernel, the virtual nodes in Mininet-BMv2 are lightweight, making it possible to achieve scalable network emulation. Although the lightweight virtual nodes in Mininet-BMv2 enable scalable network emulation, the execution order and burst lengths of virtual nodes are mostly managed independently by the host machine's operating system. This results in temporal fidelity issues, particularly in large-scale network emulation scenarios, which motivates us to explore a container-based virtual time system to provide precise time management services to Mininet-BMv2 in this work.

3 TEMPORAL ERROR ANALYSIS OF MININET-BMV2

We perform experiments to demonstrate temporal fidelity issues of Mininet-BMv2 as motivating examples for this work. The experiments were conducted on a 64-bit Linux platform (Ubuntu 20.4.5 with Linux Kernel version of 5.8.1) with two AMD EPYC 7662 CPUs and 1 TB RAM. Each experiment was repeated at least 10 times independently. We selected two distinct network topologies, including a linear topology as shown in Figure 1 and a ring topology as shown in Figure 2. Three sets of experiments were conducted to show the issues of temporal fidelity and scalability in Mininet-BMv2.

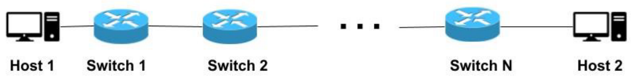


Figure 1: Linear network topology

Temporal Fidelity. Figure 3 shows the experimental results with the linear network topology consisting of two hosts and 16 switches in Mininet. The link delay was set to 1 ms using the TCLink module and the link bandwidth was configured using the traffic control (tc). We used Iperf3 to measure the TCP throughput between the two hosts. The X-axis of the figure represents the link bandwidth, while the Y-axis represents the average TCP throughput. We use both BMv2-based P4 switch and Open vSwitch (OVS) in the experiments for comparison. The orange line indicates the measurements with BMv2 and the green line indicates the measurements with OVS. The blue dashed line represents the ideal throughput (i.e., link bandwidth) under the current configuration.

The results of the experiments show that Mininet-BMv2 can maintain a good performance close to the line rate when the link bandwidth is less than 100 Mbps. However, the throughput of the testbed is at 130 Mbps when the link bandwidth reaches around

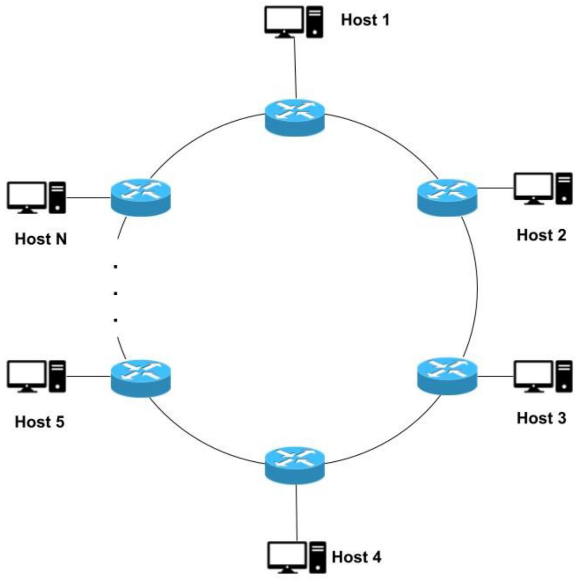


Figure 2: Ring network topology

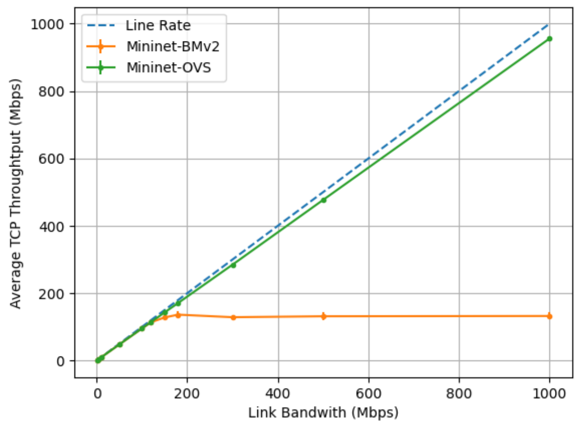


Figure 3: TCP throughput in a linear network topology with 16 switches and varying link bandwidth.

180 Mbps. The throughput remains the same even though the link bandwidth keeps increasing, resulting in a significant gap between the expected throughput and the actual measurements. Given one key advantage of the P4 switch is the high-performance packet processing speed, a saturation throughput at 130 Mbps in a small-scale emulated network scenario is not desired. In contrast, OVS can maintain a high throughput closer to the line rate as shown in Figure 3. Further exploration of the OVS performance limitation reveals that OVS can reach a throughput of up to 30 Gbps on the experimental host machine.

Various BMv2 implementations, such as simple_switch, simple_switch_grpc, psa_switch [19], have different performance (e.g., the maximum throughput ranges from 40 Mbps to 1 Gbps). The instance shown in Figure 3 uses simple_switch_grpc, and the result shows a maximum throughput of around 170 Mbps. Our experimental results using simple_switch can achieve a throughput of up to 1 Gbps. However, BMv2 still exhibits significantly lower throughput than OVS, which can be attributed to the tracing features of BMv2 that provide a rich tool for developers to debug but introduce overhead and slow down the overall performance. The experiment results indicate that turning on all tracing features could result in a performance decrease of up to 30.7%. Despite its slower performance, BMv2 remains one of the most widely used P4 software switches due to its excellent flexibility and customizability. Its compatibility with network testbeds such as Mininet also makes it easy to integrate into various testing environments. Furthermore, virtual time techniques like time dilation have shown promise in scaling BMv2's performance for high-speed P4 network scenarios. By slowing down the time advance rate, BMv2 can be scaled by multiple times, making it a valuable tool for testing and evaluating complex P4-based networks.

Scalability. We evaluated the performance of BMv2 in network topologies of varying sizes. We first conducted experiments using the Mininet-BMv2 testbed and measured the TCP throughput of a linear topology network with different numbers of switches, ranging from 1 to 256. The links in the network had a bandwidth of 500 Mbps and a delay of 1 ms. Figure 4 presents the results of our experiments. The blue dashed line represents the desired throughput, whereas the orange line shows the measured throughput. The results indicate that the throughput deviates from the desired behavior as the number of switches increases, with a significant downgrade in performance when the network scale surpasses 64 switches. For example, the network of four switches achieved an average throughput of 495.6 Mbps, which was 99.12% of the desired rate. However, the 256-switch network had a much lower throughput of 87.6 Mbps, which was only 17.5% of the desired rate. Moreover, we found that the variance of the throughput increased with the network scale.

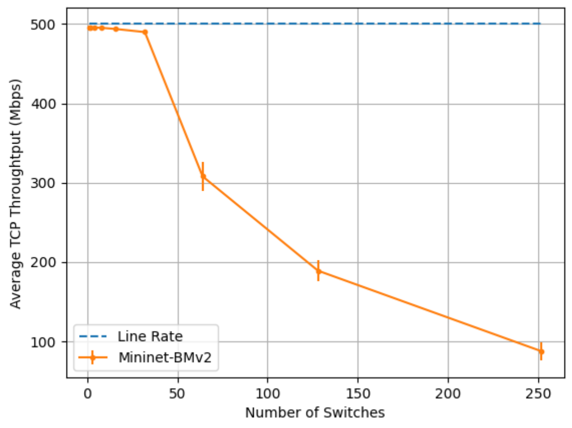


Figure 4: TCP throughput performance in a linear network topology with varying number of switches, link bandwidth fixed at 500 Mbps.

We then investigated the scalability of Mininet-BMv2 with a more complex scenario. We created a testbed with a ring topology consisting of 256 switches, where each switch was connected to a virtual host in Mininet. All links in the network had a bandwidth of 1000 Mbps and a delay of 1 ms. We split the hosts into pairs, e.g., (h1, h2), (h3, h4),...(h255, h256), and measured the throughput within each pair of hosts. We generated TCP flow between h1 and h2 for the first 40 seconds of the experiment, and then initiated TCP flow for all the other paired hosts. We expected the throughput between each pair of hosts to remain at the rate of 1000 Mbps, since the flow of each pair of hosts did not overlap with the links from the other pairs. However, the results were far below the line rate with significant disturbances. To illustrate our findings, we selected four pairs among the 256 hosts and plotted their TCP throughputs over time in Figure 5. The results show that within the first 40 seconds, the throughput between h1 and h2 remained at the desired rate of 1000 Mbps. However, after the other hosts started transmitting flows, the throughputs of all four pairs dropped significantly to around 212 Mbps, which was substantially lower than the expected rate.

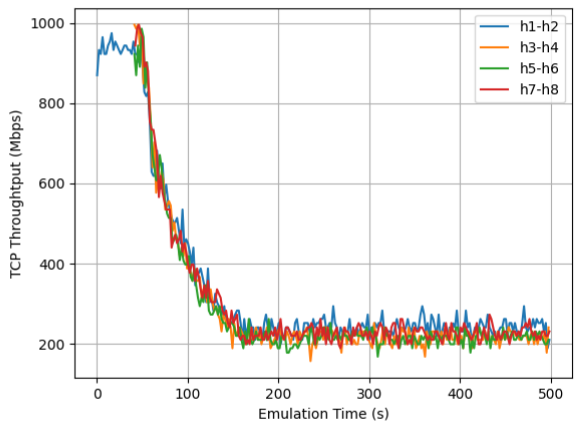


Figure 5: TCP throughput over time in a 1000-Mbps bandwidth, 256-switch ring network topology

The main cause of the inaccuracy in Mininet-BMv2 is due to the fact that containers are multiplexed on a single physical machine, sharing the same system clock and other resources of the underlying machine. Additionally, the execution order and duration of containers can vary unpredictably during an experiment, depending on the availability of physical resources (i.e. CPU and network bandwidth) on the host machine. When the number of containers exceeds the host machine's capacity, containers must race and wait for resource availability, while the system clock keeps ticking. Consequently, each container's perception of time reflects the serialization of execution on the host machine, rather than the execution of their tasks. This can cause a problem with temporal fidelity, especially when emulating large-scale networks where the host machine's resources are overwhelmed.

To overcome the issue of temporal fidelity, virtual time systems have been developed for VMs and containers used in network emulation experiments [11, 13, 16, 17, 22, 27]. Each container has its own independent virtual clock that progresses only when the container is in the execution or waiting state. Moreover, a time dilation factor (TDF) [18] is assigned to each virtual clock, which represents the ratio of wall-clock time to the perception of time in the container. For instance, a TDF of 10 means that the container experiences one second of time advancement for every ten seconds of wall-clock time, making the virtual node run at 10x the speed of the real world. By increasing the scale of interactions between containers and the underlying machine, this technique can make a 100 Mbps link appear like a 1 Gbps link from the container's perspective. In this paper, we utilize the TDF-based virtual time system to enhance the temporal fidelity of Mininet-BMv2.

4 DESIGN AND IMPLEMENTATION

We design and implement VT-BMv2, an emulation testbed that integrates virtual time systems into Mininet-BMv2 network emulator to improve performance fidelity and scalability. VT-BMv2 comprises two layers: (1) a Container-based Network Emulator integrated with a virtual time interface module, which provides each container with a virtual time perception, and (2) a Synchronization Controller that manages container execution and time synchronization. The architecture of VT-BMv2 is illustrated in Figure 6.

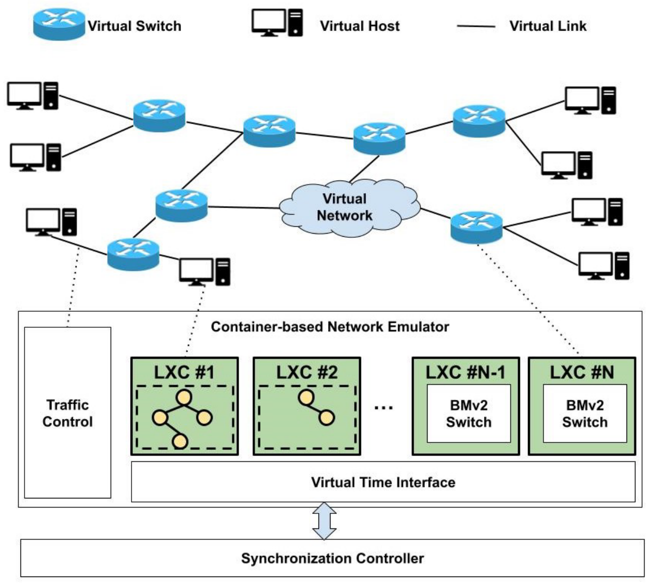


Figure 6: Architecture design of VT-BMv2

4.1 Container-based Network Emulator

A network emulation testbed typically consists of two main components: virtual nodes, such as hosts and switches, and virtual links connecting these nodes. In our approach, Container-based Network Emulator inherits from Mininet-BMv2 and enables the creation of virtual nodes using Linux containers. Each container represents a virtual host/switch within the emulation and has its own independent namespace including IP addresses and virtual network interfaces. The emulation supports multiple types of switches, such as Open vSwitch and BMv2-based P4 switches, which offer high functional fidelity through the use of compiler and runtime libraries. Virtual links, such as veth for Ethernet emulation, connect containers in the virtual network. To manage network traffic on virtual links, we use Traffic Control (tc), a Linux kernel tool that implements rules and configurations for loss, delay, link bandwidth, and traffic prioritization.

We have developed a Virtual Time Interface by making minor modifications to the Linux Kernel, such as `task_struct` structure, and `gettimeofday` function. This interface enables containers in the network emulation to maintain an independent virtual clock and a TDF, which gives each container a sense of virtual time. The interface intercepts and handles time-related functions, such as `gettimeofday` and `nanosleep`. When a process inside a container invokes a time-related system call, the Virtual Time Interface bypasses the default system call and redirects the call to a modified function that calculates the return value based on the container's virtual clock. Five fields, `isVirtual`, `virtualStartTime`, `runTime`, `TDF`, `virtualTime` are added to `task_struct` in Linux kernel (`include/linux/init_task.h`) to enable virtual clock.

- `isVirtual`: A binary flag that equals 1 when the container is integrated with virtual time.
- `TDF`: The time dilation factor that scales the virtual clock's speed relative to the physical clock.
- `virtualTime`: The time passed in the virtual clock of the container.
- `virtualStartTime`: The wall-clock time when a container is first dilated.
- `runTime`: The wall-clock time when the virtual time is last updated.

Algorithm 1 presents pseudocode for a modified Linux system call, `gettimeofday`. When called by a regular process, this function returns the result from the original system call. However, if the process is integrated with a virtual clock, the modified `gettimeofday` function calculates the run time, adjusts it with TDF, and updates the virtual time.

Algorithm 1: Virtual Time Interface

```
Function gettimeofday (lxc)
    if lxc.isVirtual == 1 then
        runTime = now - lxc.virtualStartTime;
        dilatedRunTime = (runTime - lxc.runTime)/lxc.tdf;
        virtualTime = lxc.virtualTime + dilatedRunTime;
        lxc.runTime = runTime;
        lxc.virtualTime = virtualTime;
        return virtualTime;
    else
        return do_original_gettimeofday();
    end
end
```

4.2 Synchronziation Controller

To enhance the fidelity and scalability of VT-BMv2, a barrier-based conservative synchronization module, named Synchronization Controller, is designed to manage the execution and synchronize virtual time among containers. As shown in Figure 7, the advancement of emulation is split into multiple cycles. Each cycle contains two phases: synchronization and execution. At the beginning of the synchronization phase, all containers are stopped. The Synchronization Controller interacts with the virtual time interface to extract the current state of each container and calculates the expected running time of each container in the next cycle. The execution time is carefully calculated to guarantee that the virtual time of each container is synchronized at the end of execution. The execution time is determined by two factors, containers' time dilation factor TDF and a user-defined parameter $quanta$, which defines the maximum execution time a container can obtain for one cycle. Suppose there are n containers in the emulation, denoted as LXC_i where i ranges from 1 to n. The TDF and execution time of each container LXC_i, is denoted as TDF_i and E_i, while TDF_{max} is the max TDF among all the containers. Since all the containers ought to have synchronized virtual time at the end of the cycle, we have

$$\forall 0 < i, j <= n \quad \frac{E_i}{TDF_i} = \frac{E_j}{TDF_j} \tag{1}$$

We can observe from Equation 1 that a container execution time is proportional to its TDF. The container with TDF_{max} is granted the largest execution time, $quanta$, in the current cycle. Therefore, the execution of each container is

$$\forall 0 < i <= n \quad E_i = \frac{quanta}{TDF_{max}} \times TDF_i \tag{2}$$

The synchronization controller assigns each container to a specific CPU and schedules the execution of processes within the container using a round-robin method. Note that VT-BMv2 enables the parallel execution of containers with multiple CPUs and the number of CPUs can be customized. Each container is granted a high priority when assigned to a CPU to ensure that it runs without being preempted by other processes (e.g., `sched_priority` is set to 99).

The Synchronization Controller uses a greedy scheduling algorithm to assign containers to CPUs based on the current workload of each CPU. The algorithm selects the CPU with the least workload at the current moment and assigns the container to it. Each CPU in the emulation is assigned a kernel thread (kthread), which maintains a list of containers associated with the CPU. These kthreads control the execution order and burst length of each container during the execution phase. Algorithm 2 presents the pseudocode for the scheduling algorithm.

The overhead of the algorithm is bounded by $O(m \times n)$, where m is the number of designated CPU cores and n is the number of containers. Although it may not always provide the optimal solution, using a more fine-tuned scheduling algorithm [28] could increase the time and space complexity, leading to more overhead. An alternative approach is to maintain a min heap based on the aggregated execution time of each CPU, which reduces the complexity of searching for the CPU with the minimal workload from $O(n)$ to $O(log_n)$. The design and analysis of more efficient scheduling algorithms will be considered in future work.

Algorithm 2: Schedule the Containers on Multiple CPUs

```
def scheduling ( list_kthread, list_lxc )
    // Each CPU core is assigned with a kthread
    // Each kthread maintains a linked list to
       hold containers that are to be executed on
       them in the next cycle
    for LXC in list_lxc do
        // Linearly scan all kthread to find the
           one, tk, with minimal workload
        tk;
        for ck in list_kthread do
            if tk.exe_time <= ck.exe_time then
                tk = ck;
            end
        end
        // Append container to kthread's execution
           list and update workload
        tk.list_lxc.append(lxc);
        tk.exe_time += LXC.exe_time;
    end
end
```

Once the scheduling phase is completed, the Synchronization Controller initiates the execution phase by starting the kthreads. The kthreads of each CPU wake up or pause the containers in a sequential manner based on their predetermined execution length and order, which was determined by the controller during the previous synchronization phase. It is worth noting that the execution length of a container can vary over cycles due to changes in TDF, as specified in Equation 2.

The existing virtual time systems use two primary mechanisms to control container execution: (1) a timer-based approach, such as [22], and (2) an instruction-based approach, such as x [11] and y [12]. The timer-based approach determines the execution length of a container based on the amount of real-world time elapsed, using techniques such as SIGSTOP and SIGCONT signals and high-resolution timers like hrtimer. In contrast, the instruction-based approach perceives the execution length based on the number of executed binary instructions, utilizing perf [2] to count instructions and ptrace [3] to control container execution. Algorithm 3 illustrates the detailed implementation for advancing the emulation by one cycle.

5 SYSTEM EVALUATION

In this section, we evaluate and analyze the performance of VT-BMv2 regarding performance fidelity, system overhead, and scalability. The experiments are conducted on the same Linux machine mentioned in Section 3.

Algorithm 3: Parallel Execution and Control

```
def progressForOneCycle ( list_kthread, list_lxc )
    // Schedule containers among kthread
    scheduling ( list_kthread, list_lxc );
    for kthread in list_kthread do
        // start_kthread will be executed in
           parallel
        start_kthread(kthread);
    end
    // Wait until all kthreads to complete and
       join the main process
    wait_kthreads(list_kthread);
end
def startKthread ( kthread)
    for lxc in kthread.list_lxc do
        // wakeup lxc using SIGCONT
        wakeup(lxc);
        // set a count down using highresolution
           timer
        // when time up, pause lxc using SIGSTOP
        set_hrtimer(lxc, lxc.exe_time);
        hrtimer.callback = pause(lxc);
    end
    join_main_process();
end
```

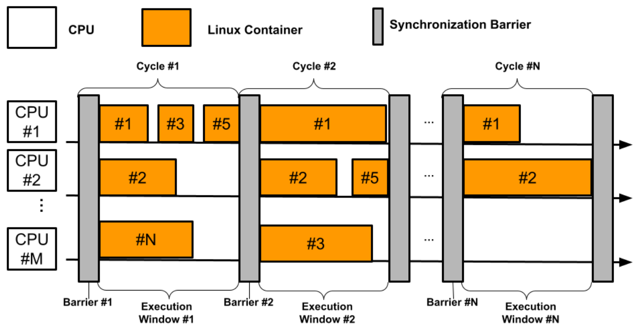
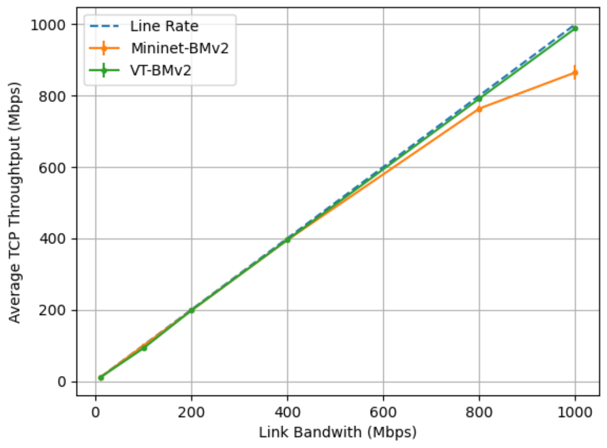


Figure 7: Barrier-based conservative synchronization

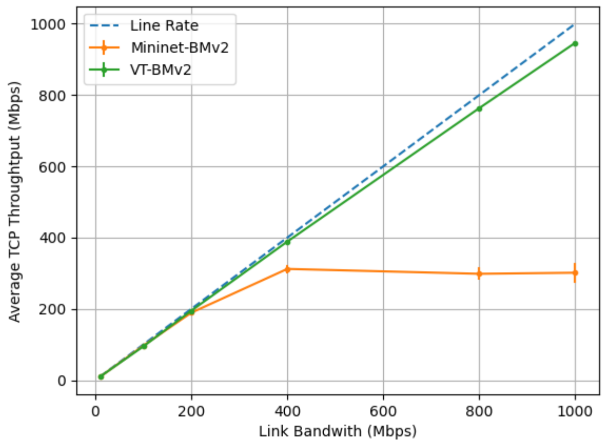
5.1 Performance Fidelity

We conducted two sets of experiments using the linear network topologies of 16 and 64 BMv2 switches on Mininet-BMv2 and VT-BMv2. The goal of the experiments is to measure TCP throughput between hosts on opposite ends of the network. The results of the experiments are shown in Figure 8, where the X-axis represents the link bandwidth and the Y-axis represents the average TCP throughput. The orange line shows measurements with Mininet-BMv2, the green line shows measurements with VT-BMv2, and the blue dashed line represents the ideal throughput under the current configuration.

The results indicate that the accuracy of VT-BMv2 outperforms Mininet-BMv2 when emulating P4 networks. Mininet-BMv2 can only maintain fidelity, i.e., close to the line rate, when the link bandwidth is relatively low, such as under 400 Mbps with 16 switches (Figure 8a) and under 200 Mbps with 64 switches (Figure 8b). However, the throughput of Mininet-BMv2 remains constant even when the link bandwidth increases, leading to a significant discrepancy between the expected and actual throughput. For example, when the link bandwidth is set to 1000 Mbps, the throughput of Mininet-BMv2 with 64 switches is only 301.5 Mbps, resulting in an error of approximately 69.8%. In contrast, VT-BMv2 maintains a consistent and accurate TCP throughput even at high link bandwidths, with a much smaller deviation from the expected throughput. For example, with a bandwidth of 1000 Mbps on the 16-switch topology, VT-BMv2 achieves a high throughput of 988.7 Mbps with a TDF of



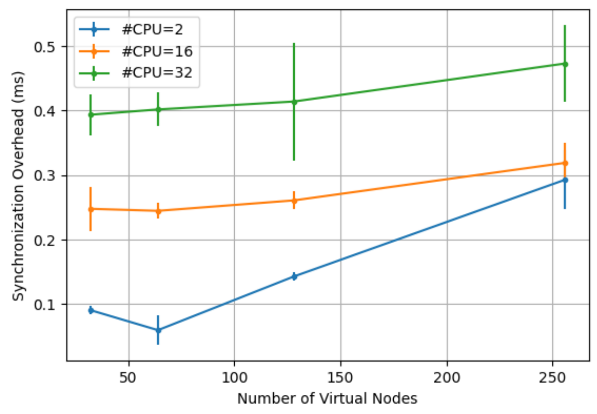
(a)



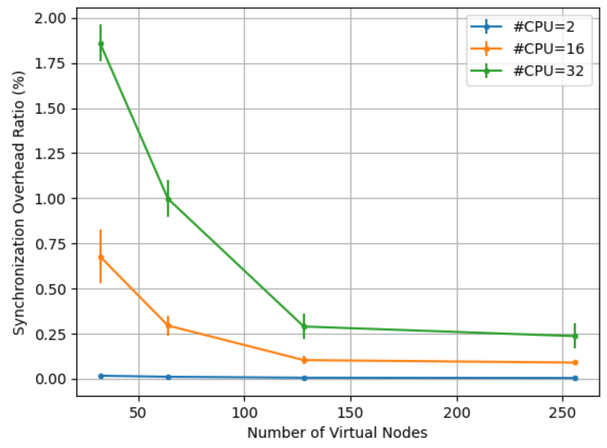
(b)

Figure 8: Comparison of TCP throughput between Mininet-BMv2 and VT-BMv2 with a link delay of 1 ms. The emulation network is constructed using a linear topology with (a) 16 and (b) 64 BMv2 switches.

5, which is only 1.13% difference from the desired bandwidth. This behavior can be attributed to the ability to effectively manage and schedule container execution in virtual time.



(a)



(b)

Figure 9: Comparison of synchronization overhead among different network sizes: (a) absolute overhead, and (b) overhead as a proportion of runtime.

5.2 Virtual Time System Overhead

In a multi-processing system, synchronization overhead typically refers to the additional time and resources needed to ensure that multiple processes can access shared resources. However, in this paper, our focus is to evaluate the additional overhead incurred by the virtual time system in the emulator. Since VT-BMv2 uses a conservative barrier-based synchronization approach, the primary overhead is incurred within the barrier. This includes the time taken by each kernel thread to complete task scheduling and the time taken to start or wait for kernel threads to join.

The relationship between the synchronization overhead and the number of nodes is depicted in Figure 9a. As the number of containers increases, the overhead grows linearly, and more containers result in a higher workload of task scheduling. However, the ratio of synchronization overhead shown in Figure 9b decreases even as the number of containers increases. This is because the execution time increases linearly with the number of containers, and the improvement in execution time is more significant than the cost of the increased overhead, resulting in performance gains with more containers. We also observe that the synchronization overhead increases as the number of CPUs increases. This is because each CPU is associated with a kernel thread, more time is needed to start or wait for threads to join, and more CPUs increase the scheduling complexity. As illustrated in Figure 9, when emulating 256 containers on 16 CPUs, the overhead introduced by VT-BMv2 is 0.31 ms with a synchronization ratio of 0.13%.

To measure the overhead of modifying the Linux kernel on system calls, we conducted experiments on the modified version of the `gettimeofday` function. The results show that the modified function introduces an additional overhead of 1304.8 nanoseconds, whereas the original call only takes 330.3 nanoseconds to return. The increased overhead is due to the need to query the previous status of the container, calculate the current virtual time, and update the container's status every time the modified system function is called.

5.3 Scalability

In order to examine the scalability of VT-BMv2, we conducted experiments on linear topology networks with varying numbers of switches, ranging from 1 to 256. The link bandwidth and delay were configured as 1000 Mbps and 1 ms, respectively. Figure 10 shows the experimental results. We observe that as the network scale increased, the throughput of Mininet-BMv2 significantly decreased, while VT-BMv2 was able to maintain a throughput close to the desired rate. In the 256-switch network, Miniet-BMv2 achieved a throughput of only 86.4 Mbps, while VT-BMv2 achieved a much higher throughput of 901.8 Mbps. By integrating the virtual time system, VT-BMv2 reduced the error rate from 91.4% in Mininet-BMv2 to just 9.8%.

To further demonstrate the improvement of VT-BMv2, we conducted another set of experiments using the same network scenario described in Figure 2 with Mininet-BMv2 results shown in Figure 5. We created a ring topology consisting of 256 switches, with each switch connected to a virtual host in Mininet. All links in the network had a bandwidth of 100 Mbps and a delay of 1 ms. We split the hosts into pairs, e.g., (h1, h2), (h3, h4),...(h255, h256), and measured the throughput between each pair. Figure 11 shows the throughput of four selected pairs. In comparison to the result shown in Figure 5, we observed that VT-BMv2 is capable of maintaining high fidelity in a large-scale P4 network with a massive volume of transmission. For example, under the same configuration, VT-BMv2 can maintain a stable TCP throughput of about 98.9% of the desired rate (1000 Mbps), which is approximately 4.6 times higher than the throughput achieved by Mininet-BMv2 shown in Figure 5.

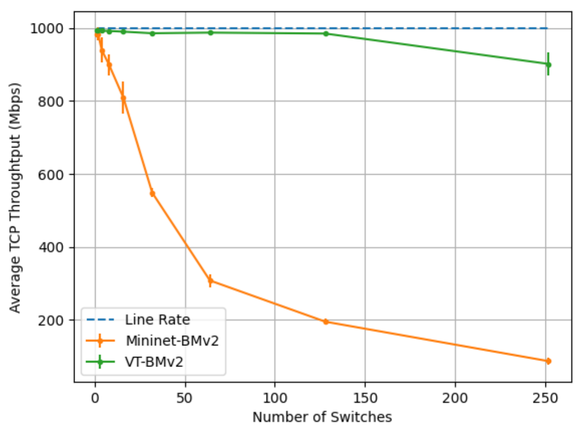


Figure 10: Scalability evaluation of Mininet-BMv2 and VT-BMv2 using linear topology networks with 1000 Mbps link bandwidth and 1 ms link delay.

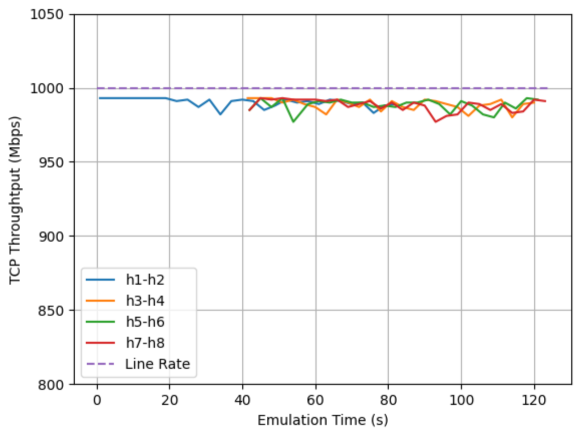


Figure 11: TCP throughput over time in a ring network topology with 256 switches and 1000 Mbps link bandwidth.

6 RELATED WORK

6.1 Virtual Time System

Several virtual time systems have been proposed based on the concept of time dilation factor (TDF), which was introduced by DieCast [17, 18] as a means of precise scaling of system capacity to match the behaviors of the target network that often exceeds available physical resources by trading time with system resources. DieCast defines TDF as the ratio of the rate at which time passes in the physical world (wall-clock time) to the perception of time by virtual machines (virtual time) [18]. Other notable virtual time systems include SVEET! [15], and TimeJails [16].

To enhance the high-fidelity and scalability of network emulation, `Anonymous` introduces a scheduling-based virtual time system for OpenVZ [1], an OS-level virtualization technology. The system assigns each process an independent virtual clock. The modified scheduler determines the execution order and burst length for each process. As a result, all virtual clocks advance in a synchronized window to enhance temporal fidelity and ensure the causality of emulated events. Other virtual time systems, such as TimeKeeper [22], are variations of this approach.

Our approach shares similarities with `Anonymous` where a lightweight virtual time system is implemented through direct kernel modification of time-related system calls in the Linux kernel. However, we are the first to apply virtual time in the context of P4 network emulation and demonstrate the effectiveness of a virtual time system in improving the performance of an early-stage model, such as BMv2, to near-production levels.

6.2 Container-based Emulation

Virtualization technologies, including Xen [13], OpenVZ [1], and Linux Containers (LXC) [20], offer isolated execution environments for running unmodified network application code on a physical machine. These environments provide experimenters with features, such as a process tree, file system, and network interfaces with IP addresses. Among these technologies, container-based virtualization, like LXC, has better performance and scalability than other options like Xen (para-virtualization) and QEMU (full-virtualization). This is because LXC allows multiple Linux instances to run on a single host while sharing the kernel, which creates less overhead than traditional virtual machines like Xen and VMWare that require separate kernels for each virtual machine.

Mininet is a commonly used network emulator that utilizes container-based virtualization techniques to construct virtual networks for network protocol and application testing. Our work involves making minor modifications to the Linux Kernel source code to enable virtual time and synchronization features within the Mininet containers. The modification details are elaborated in Section 4.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we discover and analyze the fidelity and scalability issue in Mininet-BMv2. We then design and implement a P4 emulation testbed, VT-BMv2 integrated with a virtual time system. Based on our comprehensive evaluation, Mininet-BMv2 is capable of maintaining high temporal fidelity and scalability with limited synchronization overhead. However, one drawback of using TDF to scale resources is the increased execution time. For example, setting a TDF of 2 requires emulation to run for 20 seconds to simulate a 10-second experiment. Selecting an appropriate TDF that meets network configuration requirements without significantly increasing execution time can be challenging. Our future work is to develop a TDF adapter that can dynamically adjust the TDF of each container to optimize runtime and meet configuration requirements.

ACKNOWLEDGMENTS

The authors are grateful for the support of the National Science Foundation (NSF) under Grant CNS-2247721 and the NSF Center for Infrastructure Trustworthiness in Energy Systems (CITES) under Grant EEC-2113903.

REFERENCES

[1] 2005. OpenVZ: a container-based virtualization for Linux. https://openvz.org/

[2] 2020. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page

[3] 2021. Ptrace: The linux process tracing subsystem. https://man7.org/linux/man-pages/man2/ptrace.2.html

[4] 2023. Aurora 710. https://bm-switch.com/index.php/bare-metal-switches/spine-switches/netberg-aurora710-100g-bms.html

[5] 2023. Barefoot Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html

[6] 2023. The Benefits of Programmable Switch ASICs. https://www.accton.com/Technology-Brief/benefits-programmable-switch-asics/

[7] 2023. BMv2. https://github.com/p4lang/behavioral-model

[8] 2023. Intel Tofino 2. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html

[9] 2023. Mininet Integrated with BMv2. https://github.com/p4lang/tutorials

[10] 2023. Stratum. https://opennetworking.org/stratum/

[11] Vignesh Babu and David Nicol. 2020. Precise Virtual Time Advancement for Network Emulation. In *Proceedings of the 2020 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation.*

[12] Vignesh Babu and David M. Nicol. 2018. On Repeatable Emulation in Virtual Testbeds. In *Proceedings of the 2018 Winter Simulation Conference.*

[13] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. *SIGOPS Oper. Syst. Rev.* (2003).

[14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (jul 2014), 87–95. https://doi.org/10.1145/2656877.2656890

[15] Miguel A. Erazo, Yue Li, and Jason Liu. 2009. SVEET! a scalable virtualized evaluation environment for TCP. In *Proceedings of the 5th International Conference on Testbeds and Research Infrastructures for the Development of Networks Communities and Workshops.*

[16] Andreas Grau, Steffen Maier, Klaus Herrmann, and Kurt Rothermel. 2008. Time Jails: A Hybrid Approach to Scalable Network Emulation. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation.*

[17] Diwaker Gupta, Kashi Venkatesh Vishwanath, Marvin McNett, Amin Vahdat, Ken Yocum, Alex Snoeren, and Geoffrey M. Voelker. 2011. DieCast: Testing Distributed Systems with an Accurate Scale Model. *ACM Trans. Comput. Syst.* (2011).

[18] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. 2005. To Infinity and beyond: Time Warped Network Emulation. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles.*

[19] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. 2023. A survey on data plane programming with P4: Fundamentals, advances, and applied research. *Journal of Network and Computer Applications* 212 (2023), 103561. https://doi.org/10.1016/j.jnca.2022.103561

[20] Matt Helsley. 2009. LXC: Linux container tools. https://developer.ibm.com/tutorials/l-lxc-containers/

[21] Kazumi Kumazoe, Masahiro Shibata, and Masato Tsuru. 2022. A P4 BMv2-Based Feasibility Study on a Dynamic In-Band Control Channel for SDN. In *Advances in Intelligent Networking and Collaborative Systems*, Leonard Barolli and Hiroyoshi Miwa (Eds.). Springer International Publishing, Cham, 442–451.

[22] Jereme Lamps, David M. Nicol, and Matthew Caesar. 2014. TimeKeeper: A Lightweight Virtual Time System for Linux. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation.*

[23] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* 38, 2 (mar 2008), 69–74. https://doi.org/10.1145/1355734.1355746

[24] Francesco Paolucci, Filippo Cugini, Piero Castoldi, and Tomasz Osiński. 2021. Enhancing 5G SDN/NFV Edge with P4 Data Plane Programmability. *IEEE Network* 35, 3 (2021), 154–160. https://doi.org/10.1109/MNET.021.1900599

[25] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. 2015. The Design and Implementation of Open VSwitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA) *(NSDI'15)*. USENIX Association, USA, 117–130.

[26] Brian Walters. 1999. VMware Virtual Platform. *Linux J.* (1999).

[27] Elias Weingärtner, Florian Schmidt, Hendrik Vom Lehn, Tobias Heer, and Klaus Wehrle. 2011. SliceTime: A Platform for Scalable and Accurate Network Emulation. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation.*

[28] Xuanxia Yao, Peng Geng, and Xiaojiang Du. 2013. A Task Scheduling Algorithm for Multi-core Processors. In *2013 International Conference on Parallel and Distributed Computing, Applications and Technologies.*