A Virtual Time System for Linux-container-based Emulation of Software-defined Networking

P.h.D Oral Qualifying Exam Report

Jiaqi Yan
Illinois Institute of Technology
10 West 31st Street
Chicago, IL, United States
jyan31@hawk.iit.edu

ABSTRACT

Realistic and scalable testing systems are critical to evaluate network applications and protocols to ensure successful real system deployments. Container-based network emulation is attractive because of the combination of many desired features of network simulators (scalability, low cost, reproducibility, and flexibility) and physical testbeds (high fidelity). The success of Mininet, a popular software-defined networking (SDN) emulation testbed, demonstrates the value of such approach that we can execute unmodified binary code on a large-scale emulated network with lightweight OS-level virtualization techniques.

However, an ordinary network emulator uses the system clock across all the containers even if a container is not being scheduled to run. This leads to the issue of temporal fidelity, especially with high workloads. Virtual time sheds the light on the issue of preserving temporal fidelity for large-scale emulation. The key insight is to trade time with system resources via precisely scaling the time of interactions between containers and physical devices by a factor of n, hence, making an emulated network appear to be n times faster from the viewpoints of applications in the container.

In this paper, we develop a lightweight Linux-container-based virtual time system and integrate the system to Mininet for fidelity and scalability enhancement. We also explore adaptive time dilation scheduling for balancing speed and accuracy. Experimental results demonstrate that (1) with virtual time, Mininet is able to accurately emulate a network n times larger in scale, where n is the scaling factor, with the system behaviors closely match data obtained from a physical testbed; and (2) with a threshold-based time dilation scheduling, we reduce the running time by 46% with little accuracy loss. Finally, we present a case study using the virtual-time-enabled Mininet to evaluate the limitations of equal-cost multi-path (ECMP) routing in a data center network.

1. INTRODUCTION

Researchers conducting analysis of networked computer systems are often concerned with questions of scale. What is the impact of a system if the communication delay is X times longer, the bandwidth is Y times larger, or the processing speed is Z times faster? Various testbeds have been created to explore answers to those questions before the actual deployment. Ideally, testing on an exact copy of the original system preserves the highest fidelity, but is often technically challenging and economically infeasible, especially for large-scale systems. Simulation can significantly improve the scalability and reduce the cost by modeling the real systems. However, the fidelity of modeled systems is always in question due to model abstraction and simplification. For example, large ISPs today prefer to evaluate the influence of planned changes of their internal networks through tests driven by realistic traffic traces rather than complex simulations. Network emulation is extremely useful for such scenarios, by allowing unmodified network applications being executed inside virtual machines (VMs) over controlled networking environments. This way, scalability and fidelity is well balanced as compared with physical or simulation testbeds.

A handful of network emulators have been created based on various types of virtualization technologies. Examples include DieCast [24], TimeJails [23], VENICE [34] and dONE [18], built upon full or para-virtualization (such as Xen [14]), as well as Mininet [26, 33], CORE [15] and vEmulab [28], using OS-level virtualization (such as OpenVZ [10], Linux container [4] and FreeBSD jails [2]). All those network emulators offer functional fidelity through the direct execution of unmodified code. Xen enables virtualization of different operating systems, whereas lightweight Linux container enables virtualization at the application level with two orders of magnitude more of VM (or container) instances, i.e., emulated nodes, on a single physical host, in the cost of only able to run a single type of operating system. In this work, we focus on improving the Linux container technology for scalable network emulation, in particular with the application of software-defined networks (SDN). Mininet [33] is by far the most popular network emulator used by the SDN community [20, 31, 37]. The Linux-container-based design enables Mininet users to experiment "a network in a laptop" with thousands of emulated nodes. However, Mininet cannot guarantee fidelity at high loads, in particular when the number of concurrent active events is more than the number of parallel cores. For example, on a commodity machine with 2.98 GHz CPU and 4 GB RAM providing 3 Gb/s internal bandwidth, Mininet is only capable to emulate a network up to 30 hosts, each with a 100 MHz CPU and 100 MB RAM and connected by 100 Mb/s links [26]. Emulators cannot reproduce correct behaviors of a real network with large topology and high traffic load because of the limited physical resources. In fact, the same issue occurs in many other VM-based network emulators, because a

host *serializes* the execution of multiple VMs, rather than in parallel like a physical testbed. VMs take its notion of time from the host system's clock, and hence time-stamped events generated by the VMs are multiplexed to reflect the host's serialization.

Our approach is to develop the notion of virtual time inside containers to improve fidelity and scalability of the container-based network emulation. A key insight is to trade time for system resources by precisely scaling the system's capacity to match behaviors of the target network. The idea of virtual time has been explored in the form of time-dilation-based [25] and VM-scheduling-based [40,41] designs, and has been applied to various virtualization platforms including Xen [24], OpenVZ [41], and Linux Container [32]. These related works are carefully discussed in section 6 In this work, we take a time-dilation-based approach to build a lightweight virtual time system in Linux container, and have integrated the system to Mininet for scalability and fidelity enhancement. The time dilation factor (TDF) is defined as the ratio between the rate at which wall-clock time has passed to the emulated host's perception of time [25]. A TDF of 10 means that for every ten seconds of real time, applications running in a time-dilated emulated host perceive the time advancement as one second. This way, a 100 Mbps link is scaled to a 1 Gbps link from the emulated host's viewpoint.

Our contributions are summarized as follows. First, we have developed an independent and lightweight middleware in the Linux kernel to support virtual time for Linux container. Our system transparently provides the virtual time to processes inside the containers, while returns the ordinary system time to other processes. No change is required in applications, and the integration with network emulators is easy (only slight changes in the initialization routine). Second, to the best of our knowledge, we are the first to apply virtual time in the context of SDN emulation, and have built a prototype system in Mininet. Experimental results indicate that with virtual time, Mininet is capable to precisely emulate much larger networks with high loads, approximately increased by a factor of TDF. Third, we have designed an adaptive time dilation scheme to optimize the performance tradeoff between speed and fidelity. Finally, we have demonstrated the fidelity improvement through a realistic case study about evaluation of the limitations of the equal-cost multi-path (ECMP) routing protocol in data center networks.

The remainder of the paper is structured as follows. Section 2 presents the virtual time system architecture design. Section 3 illustrates the implementation of the system and its integration with Mininet. Section 4 evaluates the virtual-time-enabled Mininet, with a case study of ECMP routing evaluation in Section 5. Section 6 discuss existing works regarding to virtual time. Section 7 concludes the paper with future works.

2. SYSTEM ARCHITECTURE DESIGN

2.1 System Overview

Figure 1 depicts the architecture of our virtual time system within a Linux-container-based network emulator. Linux container [4] is a lightweight virtualization technique that enables multiple instances of Linux OS sharing the kernel. Linux container has less overhead than full or para-virtualization platforms, such as Xen, QEMU, or VMware, in which separated kernels are required for each VM, and therefore, has been applied in the area of scalable network emulation. For example, Mininet [5] is a Linux-container-based emulation platform supporting SDN research.

Mininet creates containers to virtualize network hosts, and each container has its own private network namespace and interface. Applications (such as web services) are encapsulated in the containers. The containers are connected by software switches (typically kernel-model Open vSwitch [9]) with virtual interfaces and links as shown in Figure 1, and are multiplexed onto the physical machine. Like many other network emulators, Mininet is also vulnerable to the temporal fidelity issue in large-scale network experiments. Containers use the same system clock of the physical machine, but the execution of the containers is scheduled by the OS in serial. This leads to incorrect perception of time in a container, because the container's clock keeps advancing even if it is not running (e.g., idle, waiting, suspended). Such errors are particularly severe when emulating high workload network experiments.

To improve the temporal fidelity, we build a virtual time system as a lightweight middleware in the Linux kernel (see Figure 1). We employ the time dilation technique to provide the illusion that a container has as much processing power, disk I/O, and network bandwidth as a real physical host in a production network despite the fact that it shares the underlying resources with other containers. The basic idea is to make time appear to be slower than the wall-clock time, so that the emulated network appears faster.

A capable virtual time system for scalable network emulation needs to have the following requirements: (1) lightweight design with low system overhead, (2) transparent virtual time provision to applications in containers, i.e., no code required modification, (3) universal virtual time support within the emulator and invisible to other processes on the same machine, and (4) ease of integration to the emulator. Accurate and positive emulation results can improve the confidence that any changes (e.g., a transformation from a traditional network to an SDN-based architecture) to the target production network will be successfully deployed.

2.2 Virtual Time Management

Our virtual time system, as shown in Figure 1, is designed to meet all the requirements. The time dilation manager is responsible for computing and maintaining the virtual time, according to a given TDF for all the containers. It can offer per-container virtual time or the global virtual time for the emulator. The per-container virtual time is useful to support synchronized emulation (in virtual time) and facilitates the integration with network simulators. We have made a small set of changes in the kernel, in particular, a modified data structure to present virtual time, and a new algorithm to convert the elapsed wall-clock time to the dilated virtual time, with no dependency on third-party libraries.

We attach each container an integer-valued TDF, which could also be shared among all containers. A TDF of k slows down a container's time advancement rate by a factor of k, thus re-scales a container's notion of time with reference to a physical network. This way, Mininet can emulate a seemingly k times faster network owing to the accelerated rate of interaction between the containers and the virtual network Note that our design cannot scale the capacity of hardware

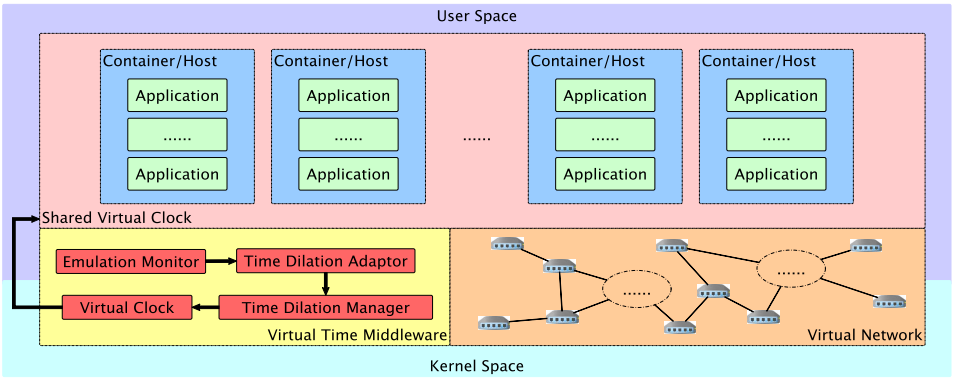


Figure 1: **Architecture of the Virtual Time System in a Container-based Network Emulator. Note that a typical container-based network emulator can be presented by this figure without the Virtual Time Middleware.**

components such as main memory, processor caches, and disk, firmware on network interfaces.

The integration with Mininet, and potentially other container-based software is straightforward. We provide a set of simple APIs to (1) initialize containers with virtual time, and (2) inquire virtual time at run time. The system returns precise virtual time to container processes and transparently to all their child processes while returning the ordinary system time to other processes. We have integrated the system with Mininet. The implementation details are discussed in Section 3, and we have madee our code base available to public on GitHub[1].

2.3 Adaptive Time Dilation

The key insight of virtual time is to trade time with available system resources. The execution time can be unnecessarily long with an overestimated TDF. It is difficult to avoid that with a fixed TDF when the resource demands vary substantially during the emulation. Therefore, we investigate means to adaptively adjust TDF in run-time with the goal of well balancing the execution speed and accuracy. We take a similar epoch-based approach described in [21], and develop two modules, Emulation Monitor and Time Dilation Adaptor (see Figure 1), to achieve the dynamic TDF adjustment.

Emulation Monitor periodically collects the process-related information (not necessarily coincides with the epoch duration) and computes the run-time emulation load statistics, such as CPU utilization, number of waiting processes, or average process waiting time. Time Dilation Adaptor takes the inputs from Emulation Monitor, and adaptively computes the TDF for the next epoch based on a heuristic algorithm, whose details are presented in Section 3. Currently we only use the CPU utilization as the feedback control indicator, and will leave the exploration of other control algorithms as future works.

3. IMPLEMENTATION

The implementation of the virtual time system and its integration with Mininet-Hifi (the latest version of Mininet) is composed of three parts, as shown in Figure 2. First, we built a lightweight and independent middleware in the Linux kernel to provide virtual time support to user-space software. Second, we slightly modified the initialization procedure of Mininet with two additional python modules to realize (adaptive) virtual time in Mininet. Third, we discuss our design to enable transparent support of virtual time for applications running in the containers.

[1]see VirtualTimeForMininet

3.1 Linux Kernel Modifications

Our implementation is based on a recent Linux kernel 3.16.3 with no third-part library dependency.

3.1.1 Timing-related Kernel Modifications

To make a process have its own perception of time, we added the following four new fields in the `task_struct` struct type.

- `virtual_start_nsec` represents the starting time that a process detaches from the system clock and uses the virtual time, in nanoseconds
- `physical_past_nsec` represents how much physical time has elapsed since the last time the process requested the current time, in nanoseconds
- `virtual_past_nsec` represents how much virtual time has elapsed since the last time the process requested the current time, in nanoseconds
- `dilation` represents the TDF of a time-dilated process

Algorithm 1 gives the details about how we implement the time dilation. To preserve an accurate virtual clock in the kernel, we added a private function `do_dilatetimeofday` in the Linux's timekeeping subsystem to keep tracking the dilated virtual time based on the physical time passed and TDF. Based on process's `virtual_start_nsec`, the system determines the type of time to return, i.e., the physical system clock time or the virtual time.

`virtual_start_nsec` in `INIT_VIRTUAL_TIME` should first be initialized to zero so that the next gettimeofday always returns the undilated time to compute and record the exact physical time that a process starts to use virtual time. To return the accurate virtual time upon requests, the duration since the last call to `do_dilatetimeofday` is calculated and precisely scaled with TDF. To enable virtual time support for a wide range of timing-related system calls, we extensively traced the routines in Linux's subsystems that request timing information (such as `getnstimeofday`, `ktime_get`,

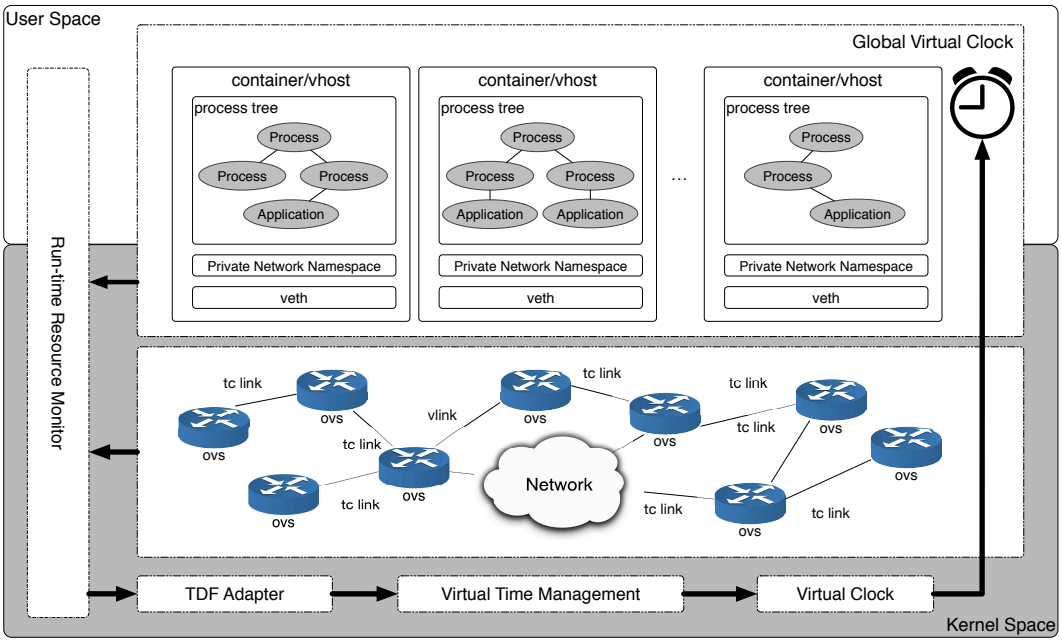


Figure 2: **Integration of Mininet-Hifi and Virtual Time**

`ktime_get_ts`, etc.), and modified them to properly invoke `do_dilatetimeofday`.

3.1.2 *Process-related Kernel Modifications*

To enable the virtual time perception to processes running in a network emulator, we added the following new system calls.

- `virtualtimeunshare` is essentially the `unshare` system call with time dilation inputs. It is used by container-based emulators, such as Mininet, to create emulated nodes. `virtualtimeunshare` creates a new process with a TDF in a different `namespace` of its parent process according to `flags`.

- `settimedilaitonfactor` offers an interface to change the TDF of a process. Note that a command executed in an emulated host is equivalent to forking a shell command executed by `bash`. Therefore, adjusting a process' TDF requires the change of the calling process' parent (e.g., host's `bash`), which occasionally would lead to tracing back to the root of the process tree, especially in the case of dynamic TDF adjustment.

We also modified the `do_fork` system call to initialize the virtual-time-related attributes of a process, such as using the variable `stack_size` to pass the TDF value. Another option is to set TDF to a default value in `virtualtimeunshare` and then relies on explicitly invoking `settimedilationfactor` to set desired TDF; this method prevents modifying the interface of creating `namespace` in traditional Linux. Functions in `timekeeping.c` were modified to invoke `do_dilatetimeofday` in order to return the virtual time to system calls like `gettimeofday` and other kernel routines that request timing information.

3.1.3 *Networking-related Kernel Modifications*

In this work, we focus on capturing all the related system calls and kernel routings to support virtual time in Linux container with the application of Mininet. One particular case related to Mininet is the usage of `tc`, a network quality-of-service control module in Linux [17]. For instance, we can use `tc` to rate-limit a link to 100 Mbps using Hierarchic Token Bucket (HTB) `qdisc` in Mininet. If the TDF is set to 8, the link bandwidth would be approximately 800 Mbps from the emulated hosts' viewpoints as we observed from the time-dilated `iperf3` application.

As a network module in Linux, `tc` does not reference Linux kernel's time as the way user application does. Therefore, `tc` is transparent to our virtual time system. One way to solve this problem is to modify the network scheduling code in kernel to provide `tc` with a dilated traffic rate. In the earlier example with TDF set to 8, the experiment will run 8 times slower than the real time, and we can configure `tc`'s rate limit as $rate/TDF = 12.5$ Mbps to emulate a 100 Mbps link. Note that we only tailored HTB in `tc`, which is the default `qdisc` used by Mininet. We will generalize the mechanism to other `qdiscs` including HFSC (Hierarchical Fair Service Curve) and TBF (Token Bucket Filter) in the future.

3.2 Network Emulator Modifications

3.2.1 *Virtual-Time-Enabled Container*

Containers allow groups of process running on the same kernel to have independent views of system resources, such as process IDs, file systems and network interfaces. We add the virtual time property to a container's `namespace` [7] so that every container can have its own virtual clock. We design our system in the way that minimal modifications of Mininet are needed for integration, so that the virtual time system can be easily extended to other Linux-container-based applications.

We modified the initialization procedure of Mininet, in particular, the `mnexec` program in Mininet, to process two

Algorithm 1 Time Dilation Algorithm

```
1: function INIT_VIRTUAL_TIME(struct task_struct *tk, int dilation)
2:     if dilation>0 then
3:         tk→virtual_start_nsec=0
4:         struct timespec ts
5:         getnstimeofday(&ts)
6:         tk→virtual_start_nsec=timespec_to_ns(&ts)
7:         tk→physical_past_nsec=0
8:         tk→virtual_past_nsec=0
9:         tk→dilation=dilation
10:    end if
11: end function
12:
13: function DO_DILATETIMEOFDAY(struct timespec *ts)
14:    Let p denote the current process using virtual time
15:    if p→virtual_start_nsec > 0 and p→dilation > 0 then
16:        now = timespec_to_ns(ts)
17:        physical_past_nsec = now - p→virtual_start_nsec
18:        virtual_past_nsec = (physical_past_nsec - p→physical_past_nsec) / p→dilation +
   p→virtual_past_nsec      /* virtual time computation */
19:        dilated_now = virtual_past_nsec + p→virtual_start_nsec
20:        dilated_ts = ns_to_timespec(dilated_now)
21:        ts→tv_sec = dilated_ts.tv_sec
22:        ts→tv_nsec = dilated_ts.tv_nsec
23:        p→physical_past_nsec = physical_past_nsec      /* update process's physical time */
24:        p→virtual_past_nsec = virtual_past_nsec      /* update process's virtual time */
25:    end if
26: end function
```

additional parameters. When we create `Node`s in Mininet (hosts, switches, and controllers are all inherited from `Node`), users can feed in a TDF argument with `virtualtimeunshare` (as a replacement of `unshare`) with `-n` option. This way, a system-wide TDF can be conveniently set for all the emulated hosts. We also provide the ability to dynamically adjust the TDF for every emulated host during runtime. To do that, we added a new option `-t` in `mnexec` to invoke the aforementioned system call `settimedilaitonfactor` to do the actual TDF adjustment. The two modifications enable the integration of virtual time in Mininet, and also serve as the basis of the adaptive TDF management.

3.2.2 Adaptive TDF Scheduling

To optimize the performance of the virtual-time-enabled Mininet, we developed an adaptive TDF scheduler through two python modules `MininetMonitor` and `DilationAdaptor` (refer to Emulation Monitor and Time Dilation Adaptor in Figure 1) to accelerate the experiment speed while preserving high fidelity.

`MininetMonitor` is responsible to monitor the CPU usage of the entire emulation system, consisting of a group of processes including the Mininet emulator, the Open vSwitch module, and emulated nodes (e.g., SDN controllers, hosts and switches). Also, applications are dynamically created, executed and destroyed within containers, in the form of child processes of their parent containers. `MininetMonitor` utilizes the `ps` command to collect the group's aggregate CPU percentage and periodically computes and passes the average CPU load statistics to `DilationAdaptor`. The core of `DilationAdaptor` is an adaptive algorithm to calculate an appropriate *TDF*. Global *TDF* updating was achieved by invoking the `mnexec -t tdf` program for every running host.

Our adaptive virtual time scheduling design is similar to the one used in [21] in spirit with two major differences. First, both techniques target on different platforms. Our technique is applied to Linux-container-based network emulation to support scalable SDN experiments, and theirs uses virtual routing and executes OpenVZ instances inside Xen. Second, their solution needs be deployed on a cluster to emulate a medium-scale network, which results in much higher communication overhead in two types: (1) every VM's monitor needs to report the CPU usage, and (2) the adaptor needs to send new TDF to every VM. Therefore, the message transmission delay in LAN and the processing delay in protocol stacks contributes to the overall communication delay. In contrast, `MininetMonitor` runs as a lightweight background thread in Mininet, and `DilationAdaptor` is simply a python object that Mininet has a reference to. The communication in our system is through synchronized queues and method invocations, which is much faster.

3.3 Virtual Time Support for Applications in Containers

Network applications running inside containers (e.g., `iperf3` [1] or `ping`) should also use virtual time. We do not need to modify the application code because they are running as child processes of Mininet's hosts. A child process always copies its parent's `task_struct` when it is forked including the same `dilation` and `virtual_start_nsec` values. Although `virtual_start_nsec` does not present the virtual starting time of the child process, our algorithm is designed to work with relative values since it does necessary initial processes during `do_fork`. When applications inquire about

the system time, the modified kernel knows that they are using virtual clock and return the virtual clock time instead of the system clock time.

One issue we notice is that the 64-bit Linux kernel running on Intel-based architectures provides the Virtual Dynamic Shared Object (vDSO) and the `vsyscall` mechanism to reduce the overhead of context switches caused by the frequently invoked system calls, for example, `gettimeofday` and `time` [8]. Therefore, applications may bypass our virtual-time-based system calls unless we explicitly use the `syscall` function. Our solution is to disable vDSO with respect to `__vdso_gettimeofday` in order to transparently offer virtual time to applications in the containers.

4. EVALUATION

In this section, we give experimental results to validate our implementation of the virtual time supported Mininet-Hifi network emulator. Very straightforward but nontrivial network typologies are adopted to demonstrate that the emulation system described in 3 can improve fidelity, scalability as well as efficiency.

All the experiments are conducted on a Dell XPS 8700 Desktop with one Intel Core i7-4790 CPU, 12 GB RAM and one gigabit Ethernet interface. The machine runs a 64-bit Ubuntu 14.04.1 LTS with our customized 3.16.3 Linux kernel. The benchmark scores of this machine's CPU and FPU are: 1.52 seconds for Blowfish, 1045.62 MiB/seconds for CryptoHash, 0.63 seconds for FFT and 2.56 seconds for Raytracing. Our virtual-time-enabled Mininet was built on the latest version of Mininet (2.1.0), also named Mininet-Hifi, at the time of development.

Fidelity. We first evaluate how our virtual time system improves Mininet's fidelity through a basic network scenario: a single TCP flow transmission through a chain of switches in an emulated SDN network. As shown in Figure 3a, the network topology consists of a client-server pair connected by a chain of Open vSwitch switches in Mininet. We setup the default OpenFlow controller to function as a learning switch. In this set of experiments, we connected two hosts through 40 switches in Mininet, and all the links are configured with 10 μs delay. We used `iperf3` [1] to generate a TCP flow between the client and the server. TDF was set to 1 (i.e., no virtual time) and 4 for comparison. We also setup a real testbed for "ground truth" throughput data collection. The testbed was composed of two machines connected by a 10 Gbps Ethernet link. We varied the bandwidth link from 100 Mbps to 10 Gbps and measured the throughput using `iperf3`. In the real testbed, we manually configured the link bandwidth and delay using `tc`, and the delay was set as the corresponding round trip times (RTTs) measured in the switch chain topology in Mininet, so that the networking settings were tightly coupled for comparison. Although we did not setup an exact network with SDN switches, the stabilized TCP throughputs generated by the physical testbed should reflect what occurs in a real SDN network. Each experiment was repeated 10 times and the results with bandwidth 4 Gbps, 8 Gbps and 10 Gbps were reported in Figure 4a.

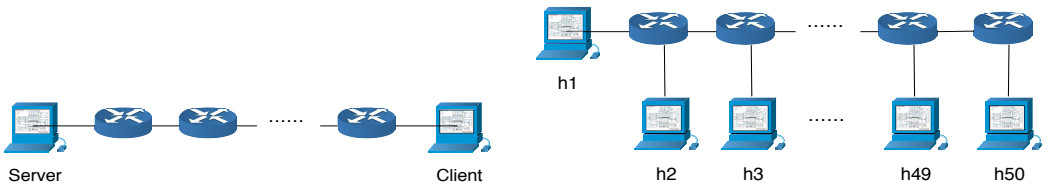
We observe that when the bandwidth was no greater than 4 Gbps (we only displayed the 4 Gbps case in the figure), Mininet was able to accurately emulate the TCP flow with and without virtual time, as the average throughputs were very close to the ones collected from the physical testbed. However, when we continued to increase the bandwidth, Mininet was not able to produce the desired throughputs, e.g., 28% (8 Gbps) and 39% (10 Gbps) smaller than the physical testbed throughputs. With virtual time (TDF = 4), Mininet was capable to accurately emulate the TCP flow even at high bandwidth, and the results were nearly the same as the ones measured in the physical testbed.

The root cause is that the machine running Mininet does not have sufficient resources to emulate networks with bandwidth greater than 4 Gbps, which would lead to fidelity issues, e.g., low expected throughputs. Note that we only emulated a single flow, and the results could be much worse and unpredictable in complicated multi-flow scenarios. Results show that virtual time can significantly enhance the performance fidelity by "slowing down" the experiments so that the system has sufficient resources and time to correctly process the packets. We further illustrate the effect by plotting the time series of throughput changes for the 10 Gbps cases in Figure 4b. With virtual time, the throughputs measured in Mininet closely match the real testbed results; without virtual time ($TDF = 1$), the ramp up speed was much slower, in particular, 22 seconds ($TDF = 1$) rather than 4 seconds ($TDF = 4$), and the throughput was incorrectly stabilized below 6.1 Gbps.

Scalability. Virtual time also improves the scale of networks that one can emulate without losing fidelity. In this set of experiments, we used the same switch chain topology in Figure 3a, and set the link bandwidth to 4 Gbps. We want to investigate, with virtual time, how many switches Mininet is capable to emulate without losing fidelity, i.e., preserving nearly 4 Gbps throughput. This time we increased the number of switches with the following values `{20, 40, 60, 80, 100}`, and TDF was selected to be 1 (no virtual time) and 4. We ran `iperf3` for 25 seconds between the client and the server. Each experiment was repeated ten times, and the throughput measurement is reported in Figure 5a.

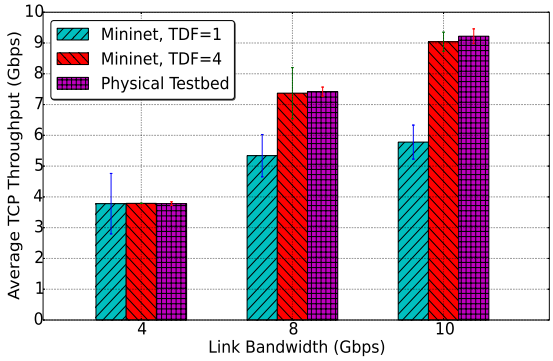
In the case of $TDF = 1$, the average throughput kept decreasing as the number of switches grew over 20. The throughput decreased dramatically when the number of switches was greater than 60 (e.g., decreased by 41% for 60 switches, 65% for 80 switches, and 83% for 100 switches). The standard deviation, indicated the undesired high disturbance, also grew as number of switches increased. When virtual time was applied with $TDF = 4$, the throughput was always around 3.8 Gbps with small standard derivations in all the experiments. It is clear that virtual time helps to scale up the emulation. In this case, Mininet can emulate 100 switches with 4 Gbps links and still generate the accurate throughputs, rather than being saturated at 20 switches without virtual time.

We also recorded the running time in Figure 5b. Longer execution time is the tradeoff for the fidelity and scalability improvement. When $TDF = 4$, the execution time was about 4 times longer than the time required in the case of $TDF = 1$ in all the experiments. In fact, we have conducted extensive experiments with different TDF values on multiple network scenarios. The general observation is that a larger TDF allows an emulator to conduct accurate experiments with larger scale on the same physical machine, but typically requires longer execution time, approximately proportional to the TDF. This leads to the question on how to balance the speed and fidelity, and our approach is to

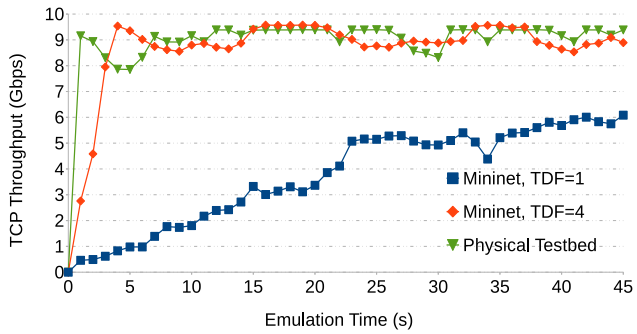


(a) **Switch Chain Topology for Fidelity and Scalability Evaluation** (b) **Network Topology for Adaptive Virtual Time Evaluation**

Figure 3: **Network Topologies for Evaluation**

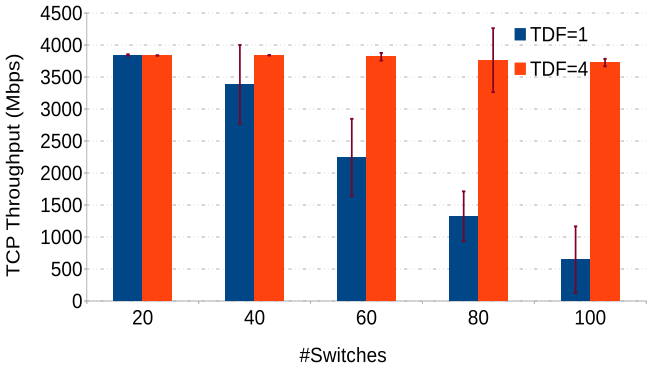


(a) **TCP Throughput with Different Link Bandwidth**

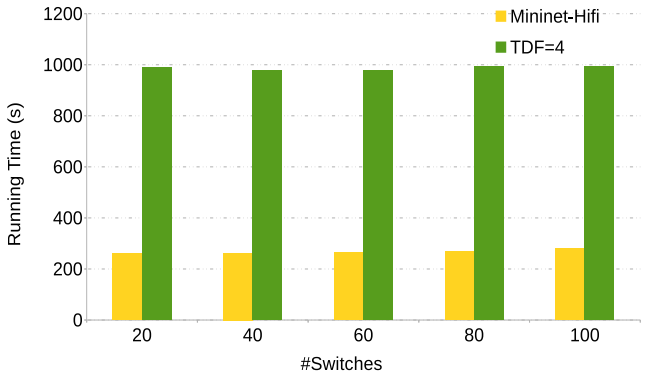


(b) **TCP Throughput with 10 Gbps Links**

Figure 4: **Fidelity Evaluation Experimental Results**



(a) **TCP Throughput**



(b) **Emulation Running Time**

Figure 5: **Scalability Evaluation Experimental Results**

explore the adaptive time dilation scheduling, whose evaluation is presented in the next section.

Adaptive TDF Scheduling. We design a set of emulation experiments consisting of multiple TCP flows to evaluate the adaptive TDF scheduling algorithm. The network topology has a simple linear structure as shown in Figure 3b and consists of 100 hosts and 99 switches. All the links are of 100 Mbps bandwidth and 1 ms delay. We selected 5 non-overlap client-server pairs: `(h1, h20), (h21, h40), h(41, h60), h(61, h80), (h81, h100)`. The entire experiment was divided into three phases: (1) initially, transmit flow `(h1,h20)`, (2) after 50 seconds, transmit all five flows, and (3) after 150 seconds, stop all the transmissions except for flow `(h1,h20)`. The goal is to evaluate how our adaptive time dilation scheduler behaves under dynamic emulation workloads with the peak load exceeding Mininet's capability.

We ran the experiments in three cases. In case 1, TDF was set to 1 (i.e., no virtual time) and the adaptive virtual time scheduling was disabled. All flows' TCP throughputs measured by `iperf3` over time are plotted in Figure 6a. In case 2, we enabled the adaptive time dilation management system with TDF initially set to 1, and conducted the same emulation experiments. Figure 6b plots the throughputs of all five flows. In case 3, we used a fixed TDF ($TDF = 11$) and disabled the adaptive virtual time scheduling. Results are shown in Figure 6c. We set $TDF = 11$ because 11 was the largest value observed in the TDF changing history in case 2. In addition, the entire trace of the dynamic TDF in case 2 is plotted in Figure 6d. We repeated each experiment for 5 times and observed very similar behaviors. All the time series reported in Figure 6 were based on the data collected from one run.

In phase 1, Mininet had sufficient system resources to emulate a single TCP flow `(h1, h21)`. Therefore, we observe the close-to-line-rate throughput, i.e., 100 Mbps, in all three cases. In phase 2, there were five concurrent flows in the network and each case demonstrated different behaviors. Note that those flows were non-overlap flows because they did not share any links or switches. Therefore, all five flows should achieve close-to-line-rate throughputs, i.e., 100 Mbps, in physical world applications. In case 1, the throughputs of all five flows were very unstable as shown in Figure 6a, which reflected the heavy resource contention in Mininet. In contrast, in case 3, all five flows have stable, close-to-100-Mbps throughputs because of the virtual time. In case 2, we observed disturbances in throughput at the beginning of phase 2, but the five flows quickly converged to the stable close-to-line-rate throughput because the adaptive TDF scheduler managed to compute the optimal TDF value. The details of TDF adjustment are depicted in Figure 6d. In phase 3, the emulation returned back to a single flow `(h1, h21)`, and the measured throughputs were accurate in all three cases. As indicated by Figure 6d, our scheduler decreased the TDF value accordingly in case 2 to save emulation time in phase 3 while still preserving the fidelity.

Table 1 summarizes the execution time, the average TDF, and the rate of execution time in wall clock to the emulation time (200 seconds) of all three cases. We can see that case 3 ($TDF = 11$) is around 10 times slower than case 1 ($TDF = 1$) in order to guarantee fidelity. Our adaptive time dilation scheduler managed to reduce 46% of the running time as compared to case 3 with little fidelity loss.

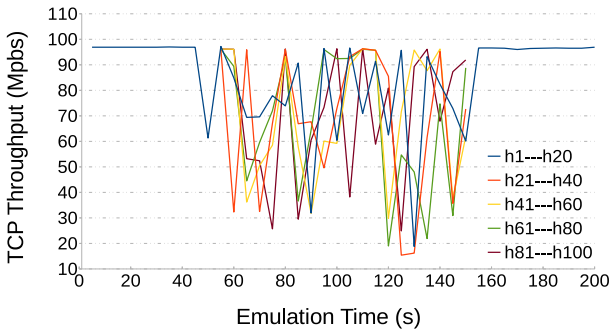
System Overhead. Our virtual time system introduces overhead with the following two reasons: (1) the computation cost in Algorithm 1 and (2) the pauses of emulation when changing containers' TDFs. We measured both types of overhead and report the results in Table 2.

First, we invoked both non-dilated and dilated `gettimeofday` 10,000,000 times from a user space application. The average overhead for one dilated `gettimeofday` is 0.013 microseconds. We then used `strace` to count the number of invocations for `gettimeofday` in a 60-second `iperf3` run on both the server and the client. The total overhead is 18,145 microseconds after tracing 1,397,829 calls, which is about 0.03% of the 60-second experiment. Actually, `iperf3` intensively invokes `gettimeofday`, because its timer is designed to exhaustively inquiry the OS time. The overhead amount will be even less for many other network applications. We also repeatedly changed a process's TDF 10,000,000 times using another test program. The average pause time was 0.063 microseconds, which is reasonably small. Since the number of TDF changes issued by the current adaptive TDF scheduling algorithm is a few orders of magnitude less than the number of calls to `gettimeofday` (e.g., only 14 TDF transitions occurred per host over the period of 1,332 seconds in the earlier adaptive TDF experiment), that overhead is also negligible.

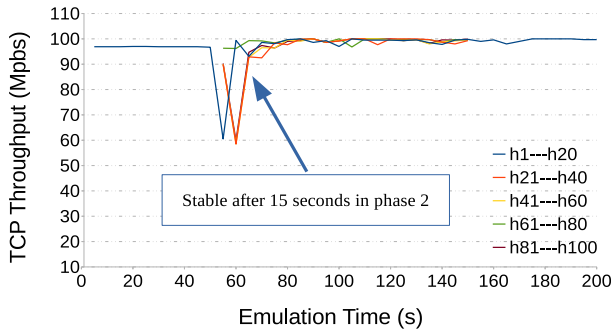
5. CASE STUDY: EVALUATION OF ECMP IN DATA CENTER NETWORKS

Network emulation testbeds are widely used to test and evaluate designs of network applications and protocols with the goal of discovering design limitations and implementation faults before the real system deployment. In this section, we present a case study to demonstrate how our virtual-time-enabled Mininet has been utilized to reproduce and validate the limitations of the equal-cost multi-path (ECMP) routing strategy in a data center network.

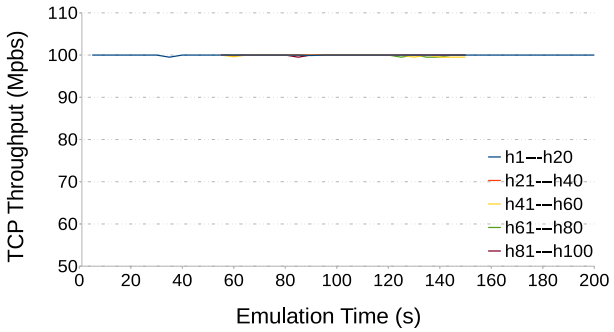
Many modern data center networks employ multi-rooted hierarchical tree topologies, and therefore ECMP-based protocols [6] are commonly used in data center networks for load-balancing traffic over multiple paths. When an ECMP-enabled switch has multiple next-hops on the best paths to a single destination, it selects the forwarding path by performing a modulo-N hash over the selected fields of a packet's header to ensure per-flow consistency. The key limitation of ECMP is that the communication bottleneck would occur when several large and long-lived flows collide on their hash and being forwarded to the same output port [16]. We borrowed the experiment scenario on a physical testbed described in [16], and created a set of emulation experiments in Mininet to demonstrate the limitation of ECMP. We built a fat-tree topology in Mininet as shown in Figure 7, and generated stride-style traffic patterns. Note that stride(i) means that a host with index x sends to the host with index $(x+i)$ mod n, where n is the number of hosts [16]. The hash-based ECMP mechanism is provided by the RipL-POX SDN controller [12]. The Mininet code was developed with reference to [11]. In all the following experiments, we set up 8 sender-receiver pairs transmitting stride-pattern traffic flows using step 1 and 4. Figure 8 shows the worst-case collision of 2 flows, distinguished by color, when stride step is 1; while in Figure 9, we see that 2 TCP flows will cause a lot



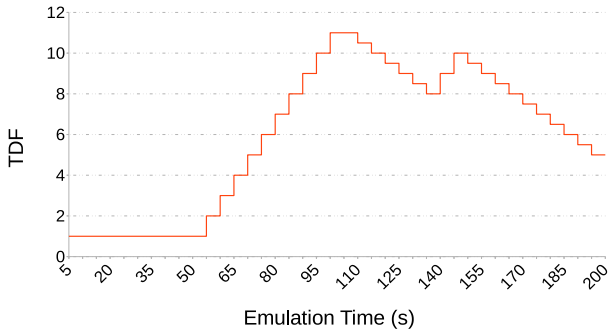
(a) **TCP Throughput without Virtual Time**



(b) **TCP Throughput with Adaptive Time Dilation**



(c) **TCP Throughput with Fixed Time Dilation (11)**



(d) **TDF Trace: Adaptive Virtual Time Scheduling**

Figure 6: **Experimental Results: Adaptive Virtual Time Scheduling Evaluation**

Table 1: **Comparison of Emulation Execution Time**

	No Virtual Time	Adaptive Virtual Time	Fixed Virtual Time
Running Time (s)	240.730	1332.242	2434.910
Average TDF	1.000	5.900	11.000
Slow Down Ratio	1.000	5.534	10.115

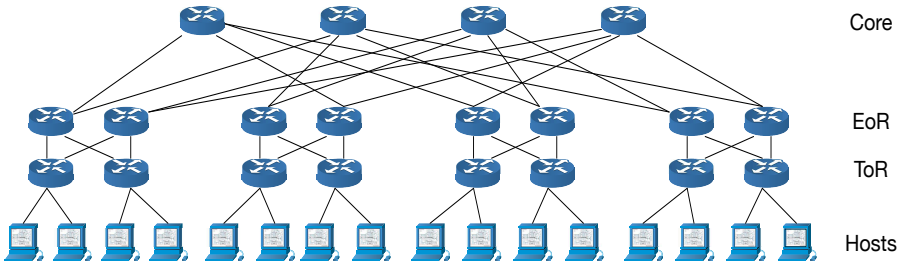


Figure 7: **A Data Center Network with a Degree-4 Fat Tree Topology**

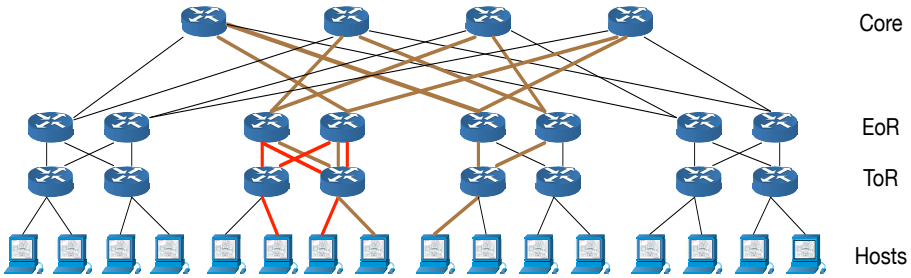


Figure 8: **Worst-case TCP Flows with Stride Step = 1**

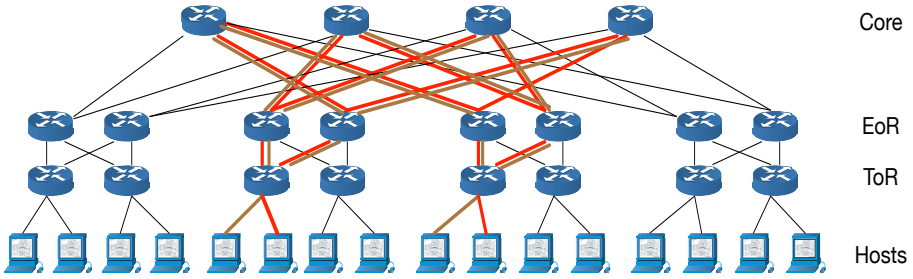


Figure 9: **Common case TCP Flows with Stride Step = 4**

of collision between core layer and aggregate layer, which is very common case.

We first set all the link bandwidth (switch-to-switch and switch-to-host) to 100 Mbps, and conducted each experiment over three independent runs. The average throughput of 8 TCP flows was plotted in Figure 10a, and each individual flow's throughput (24 in total) was plotted in Figure 10b. The limitation of ECMP presented in [16] was clearly observed. When many conflicting flows occurred with stride-4 flow patterns, the average throughput in the fat-tree network dramatically fell below 30 Mbps with up to 75% throughput drop. As shown in Figure 10b, every flow's throughput was largely affected by the hash collision limitation of ECMP in the stride-4 scenario.

However, the link bandwidth configuration in the previous experiments are not realistic. As early as in 2009, links connecting edge hosts to top of rack switches (ToR), ToR to edge of rank switches (EoR), and EoR to Core switches in a data center had been already above gigabit, in particular, 10 Gbps switch-to-switch links and 1 Gbps host-to-switch links [38]. Can Mininet still show us the limitation of ECMP with such high link bandwidth? If not, can virtual time help to overcome the issue? Using the same configurations except that links were set to 10 Gbps, we re-ran the experiments in Mininet without virtual time ($TDF = 1$) and with virtual time ($TDF = 4$). We plotted the average flow throughput in Figure 11a and individual flow throughput in Figure 11b.

In the case of stride-1, there were very few collisions among flows. Hence, the network performance ought to be close to the ideal throughput, i.e., 160 Gbps bisection bandwidth and 10 Gbps average bandwidth between each pair. In the experiments that $TDF = 4$, the average throughput is above 9.0 Gbps, which is close to the theoretical value, and also match well with the results obtained from the physical testbed built upon 37 machines [16]. In the experiments that $TDF = 1$, however, the throughput barely reaches 3.8 Gbps because of the limited system resources that Mininet can utilize. In addition, as shown in Figure 11b, we observe that the variation of throughput is large among flows when $TDF = 1$. This is incorrect because no flow shares the same link in the case of stride-1. In contrast, when $TDF = 4$, the throughput of all 8 flows are close with little variation, which implies the desired networking behaviors.

In the case of stride-4, flows may collide both on the upstream and the downstream paths, thus using ECMP could undergo a significant throughput drop, e.g., up to 61% as experimentally evaluated in [16]. The virtual-time-enabled Mininet ($TDF = 4$) has successfully captured such throughput drop phenomenon. We can see that average throughput dropped about 80% when RipL-Pox controller used ECMP to handle multi-path flows. Large deviation (more than 55% of average throughput value) also indicates that the flows were not properly scheduled with ECMP. When $TDF = 1$ (no virtual time), Mininet also reported plummeted TCP throughput in the case of stride-4. However, we cannot use the result to experimentally demonstrate the limitation of ECMP. It is hard to distinguish whether the throughput drop was caused by insufficient resources to handle 10 Gbps or the limitation of ECMP, given the fact that the throughput was already too low in the collision-free case. Without a correct baseline (benchmark for the collision-free scenario), it is difficult to perform further analysis and qualify the limitation.

6. RELATED WORK

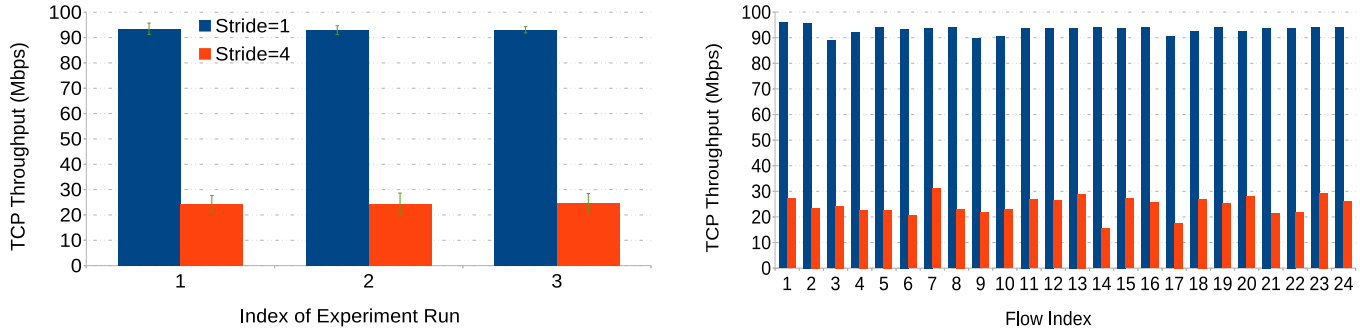
6.1 Virtual Time System

Virtual time has been investigated to improve the scalability and fidelity of virtual-machine-based network emulation. There are two main approaches to develop virtual time systems. The first approach is based on time dilation, a technique to uniformly scale the virtual machine's perception of time by a specified factor. It was first introduced by Gupta et al. [25], and has been adopted to various types of virtualization techniques and integrated with a handful of network emulators. Examples include DieCast [24], SVEET [19], NETbalance [22], TimeJails [23, 35] and TimeKeeper [32]. The second approach focuses on synchronized virtual time by modifying the hypervisor scheduling mechanism. Hybrid testing systems that integrate network emulation and simulation have adopted this approach. For example, [30] integrates an OpenVZ-based virtual time system [41] with a parallel discrete-event network simulator by virtual timer. SliceTime [40] integrates ns-3 [27] with Xen to build a scalable and accurate network testbed.

Our approach is technically closest to TimeKeeper [32] through direct kernel modifications of time-related system calls. The differences are (1) we are the first to apply virtual time in the context of SDN emulation, (2) our system has a wider coverage of system calls interacting in virtual time, and (3) our system has an adaptive time dilation scheduler to balance speed and fidelity for emulation experiments.

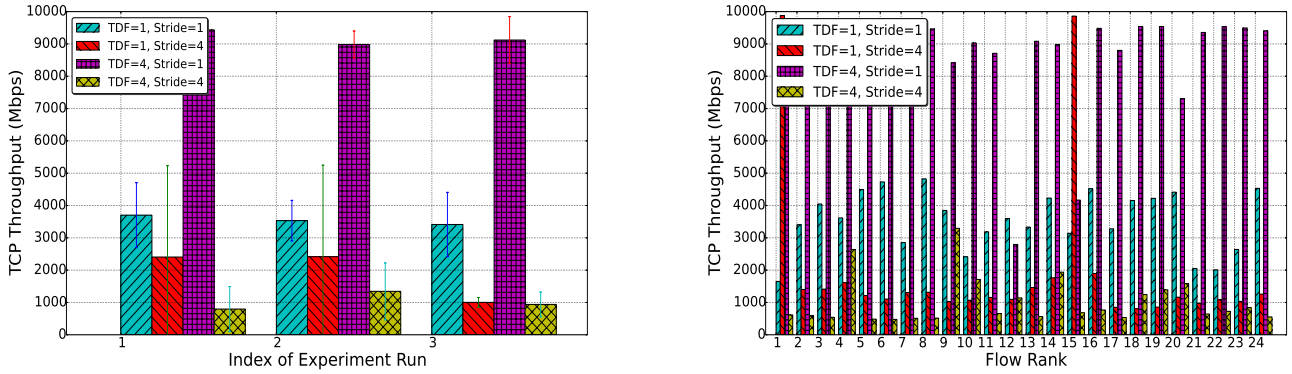
6.2 Adaptive Virtual Time Scheduling

The key insight of virtual time is to trade time with system resources. Therefore, a primary drawback is the proportionally increased execution time. To determine an appropriate TDF, VENICE [34] proposes a static management scheme to forecast the recourse demand. One problem of static time



(a) **Average TCP Flow Throughput** (b) **Throughput of Individual TCP Flow**

Figure 10: **Mininet Emulation Results: ECMP Limitation in a Fat-tree-based Data Center Network with 100 Mbps Link Bandwidth**



(a) **Average TCP Flow Throughput** (b) **Throughput of Individual TCP Flow**

Figure 11: **Mininet Emulation Results with Virtual Time: ECMP Limitation in a Fat-tree-based Data Center Network with 10 Gbps Link Bandwidth**

Table 2: **Lightweight Virtual Time System: Overhead of System Calls**

	No Virtual Time	Virtual Time	Avg Overhead Per System Call
`gettimeofday`	0.0532 μs	0.0661 μs	0.0129 μs
`settimedilationfactor`	0	0.0628 μs	0.0628 μs

dilation management is that we often assume the maximum load to ensure fidelity and thus overestimate the scaling factor.

TimeJails [23] presents a dynamic management scheme [21] to adjust TDF in run-time based on CPU utilization. We take a similar approach with two differences: the target platform and communication overhead. TimeJails is a Xen-based platform extended to a 64-node cluster for scalability, while our system supports more scalable experiments on a single machine with Linux container. TimeJails requires a special protocol to prioritizing TDF request message in local area networks, while the communication overhead of our system is much lower, either through synchronized queues or method invocations among extended modules in the emulator.

6.3 SDN Emulation and Simulation

OpenFlow [36] is the first standard communications interface defined between the control and forwarding planes of an SDN architecture. Examples of OpenFlow-based SDN emulation testbeds include Mininet [33], Mininet-HiFi [26], EstiNet [39], ns-3 [27] and S3FNet [29]. Mininet is currently the most popular SDN emulator, which uses process-based virtualization technique to provide a lightweight and inexpensive testbed. NS-3 [27] has an OpenFlow simulation model and also offers a realistic OpenFlow environment through its generic emulation capability, which has been linked with Mininet [3]. S3FNet [29] supports scalable simulation/emulation of OpenFlow-based SDN through a hybrid platform that integrates a parallel network simulator and a virtual-time-enabled OpenVZ-based network emulator [13].

7. CONCLUSION AND FUTURE WORKS

In conclusion, we present a Linux-container-based virtual time system and integrated it to a widely used SDN emulator, Mininet. The lightweight system uses a time-dilation-based design to offer virtual time to the containers, as well as the applications running inside the containers with no code modification. Experimental results show the promising fidelity and scalability improvement of Mininet with virtual time, particularly for high workload network scenarios. We have also used the platform to precisely evaluate the limitations of the ECMP routing in a realistic data center network with the results being validated by a physical testbed. Future works include the investigation of other effective control algorithms to further improve the adaptive TDF scheduler, and the integration to network simulators based on virtual time for large-scale network analysis.

8. REFERENCES

[1] iperf3. `http://software.es.net/iperf`. [Last accessed December 2014].

[2] Jails under FreeBSD 6. `http://www.freebsddiary.org/jail-6.php`. [Last accessed December 2014].

[3] Link modeling using ns 3. `https://github.com/mininet/mininet/wiki`. [Last accessed April 2015].

[4] Linux containers. `https://linuxcontainers.org`. [Last accessed December 2014].

[5] Mininet: An instant virtual network on your laptop (or other PC). `http://mininet.org/`. [Last accessed Noverber, 2014].

[6] Multipath issues in unicast and multicast next-hop selection. `https://tools.ietf.org/html/rfc2991`. [Last accessed March 2015].

[7] Namespaces in operation, part 1: namespaces overview. `http://lwn.net/Articles/531114`. [Last accessed December 2014].

[8] On vsyscalls and the vDSO. `http://lwn.net/Articles/446528/`. [Last accessed March 2015].

[9] Open vSwitch. `http://openvswitch.org`. [Last accessed Noverber 2014].

[10] OpenVZ linux container. `http://openvz.org/Main_Page`. [Last accessed October 2014].

[11] Reproducing network research. `https://reproducingnetworkresearch.wordpress.com`. [Last accessed March 2015].

[12] RipL-POX (Ripcord-Lite for POX): a simple network controller for OpenFlow-based data centers. `https://github.com/brandonheller/riplpox`. [Last accessed March 2015].

[13] S3F/S3FNet: Simpler scalable simulation framework. University of Illinois at Urbana-Champaign. `https://s3f.iti.illinois.edu/`. [Last accessed Noverber 2014].

[14] The Xen project. `http://www.xenproject.org`. [Last accessed Noverber 2014].

[15] J. Ahrenholz, C. Danilov, T. Henderson, and J. Kim. Core: A real-time network emulator. In *Proceedings of the 2008 IEEE Military Communications Conference*, pages 1–7, Washington, DC, USA, Nov 2008. IEEE Computer Society.

[16] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, pages 19–33, Berkeley, CA, USA, April 2010. USENIX Association.

[17] W. Almesberger. Linux traffic control: implementation overview. In *Proceedings of 5th Annual Linux Expo*, pages 153–164, April 1999.

[18] C. Bergstrom, S. Varadarajan, and G. Back. The distributed open network emulator: Using relativistic time for distributed scalable simulation. In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*, pages 19–28, Washington, DC, USA, May 2006. IEEE Computer Society.

[19] M. Erazo, Y. Li, and J. Liu. Sveet! a scalable virtualized evaluation environment for tcp. In *Proceedings of the 2009 Testbeds and Research Infrastructures for the Development of Networks Communities and Workshops*, pages 1–10, Washington, DC, USA, April 2009. IEEE Computer Society.

[20] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *Proceedings of the 16th ACM SIGPLAN International Conference on*

Functional Programming, pages 279–291, New York, NY, USA, September 2011. ACM.

[21] A. Grau, K. Herrmann, and K. Rothermel. Efficient and scalable network emulation using adaptive virtual time. In *Proceedings of the 18th International Conference on Computer Communications and Networks*, pages 1–6, Washington, DC, USA, August 2009. IEEE Computer Society.

[22] A. Grau, K. Herrmann, and K. Rothermel. Netbalance: Reducing the runtime of network emulation using live migration. In *Proceedings of the 20th International Conference on Computer Communications and Networks*, pages 1–6, Washington, DC, USA, July 2011. IEEE Computer Society.

[23] A. Grau, S. Maier, K. Herrmann, and K. Rothermel. Time jails: A hybrid approach to scalable network emulation. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, pages 7–14, Washington, DC, USA, June 2008. IEEE Computer Society.

[24] D. Gupta, K. V. Vishwanath, M. McNett, A. Vahdat, K. Yocum, A. Snoeren, and G. M. Voelker. Diecast: Testing distributed systems with an accurate scale model. *ACM Transactions on Computer Systems*, 29(2):1–48, may 2011.

[25] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker. To infinity and beyond: Time warped network emulation. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 1–2, New York, NY, USA, October 2005. ACM.

[26] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, pages 253–264, New York, NY, USA, December 2012. ACM.

[27] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena. Network simulations with the ns-3 simulator. *SIGCOMM Demonstration*, 15:17, August 2008.

[28] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale virtualization in the emulab network testbed. In *Proceedings of the USENIX 2008 Annual Technical Conference*, pages 113–128, Berkeley, CA, USA, June 2008. USENIX Association.

[29] D. Jin and D. M. Nicol. Parallel simulation of software defined networks. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 91–102, New York, NY, USA, May 2013. ACM.

[30] D. Jin, Y. Zheng, H. Zhu, D. M. Nicol, and L. Winterrowd. Virtual time integration of emulation and parallel simulation. In *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, pages 201–210, Washington, DC, USA, May 2012. IEEE Computer Society.

[31] E. Keller, S. Ghorbani, M. Caesar, and J. Rexford. Live migration of an entire network (and its hosts). In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 109–114, New York, NY, USA, October 2012. ACM.

[32] J. Lamps, D. M. Nicol, and M. Caesar. Timekeeper: A lightweight virtual time system for linux. In *Proceedings of the 2nd ACM SIGSIM/PADS Conference on Principles of Advanced Discrete Simulation*, pages 179–186, New York, NY, USA, May 2014. ACM.

[33] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6, New York, NY, USA, May 2010. ACM.

[34] J. Liu, R. Rangaswami, and M. Zhao. Model-driven network emulation with virtual time machine. In *Proceedings of the Winter Simulation Conference*, pages 688–696, Washington, DC, USA, December 2010. IEEE Computer Society.

[35] S. Maier, A. Grau, H. Weinschrott, and K. Rothermel. Scalable network emulation: A comparison of virtual routing and virtual machines. In *Proceedings of 12th IEEE Symposium on Computers and Communications*, pages 395–402, Washington, DC, USA, July 2007. IEEE Computer Society.

[36] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, April 2008.

[37] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. *ACM SIGCOMM Computer Communication Review*, 42(4):323–334, August 2012.

[38] A. Vahdat. Scale and efficiency in data center networks. Technical report, UC San Diego, 2009.

[39] S.-Y. Wang, C.-L. Chou, and C.-M. Yang. Estinet openflow network simulator and emulator. *Communications Magazine, IEEE*, 51(9):110–117, September 2013.

[40] E. Weingärtner, F. Schmidt, H. V. Lehn, T. Heer, and K. Wehrle. Slicetime: A platform for scalable and accurate network emulation. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, pages 253–266, Berkeley, CA, USA, March 2011. USENIX Association.

[41] Y. Zheng and D. M. Nicol. A virtual time system for openvz-based network emulations. In *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*, pages 1–10, Washington, DC, USA, June 2011. IEEE Computer Society.