VT-IO: A Virtual Time System Enabling High-fidelity Container-based Network Emulation for I/O Intensive Applications

GONG CHEN, Illinois Institute of Technology, USA
ZHENG HU, University of Arkansas, USA
YANFENG QU, University of Arkansas, USA
DONG JIN, University of Arkansas, USA

Network emulation allows unmodified code execution on lightweight containers to enable accurate and scalable networked application testing. However, such testbeds cannot guarantee fidelity under high workloads, especially when many processes concurrently request resources (e.g., CPU, disk I/O, GPU, and network bandwidth) that are more than the underlying physical machine can offer. A virtual time system enables the emulated hosts to maintain their own notion of virtual time. A container can stop advancing its time when not running (e.g., in an idle or suspended state). The existing virtual time systems focus on precise time management for CPU-intensive applications but are not designed to handle other operations, such as disk I/O, network I/O, and GPU computation. In this paper, we develop a lightweight virtual time system that integrates precise I/O time for container-based network emulation. We model and analyze the temporal error during I/O operations and develop a barrier-based time compensation mechanism in the Linux kernel. We also design and implement Dynamic Load Monitor (DLM) to mitigate the temporal error during I/O resource contention. VT-IO enables accurate virtual time advancement with precise I/O time measurement and compensation. The experimental results demonstrate a significant improvement in temporal error with the introduction of DLM. The temporal error is reduced from 7.889 seconds to 0.074 seconds when utilizing the DLM in the virtual time system. Remarkably, this improvement is achieved with an overall overhead of only 1.36% of the total execution time.

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

Additional Key Words and Phrases: Virtual Time, Network Emulation, Linux Container, TimeKeeper, Virtualization

1 INTRODUCTION

Testing networked applications is a critical but challenging problem because today's networks experience enormous dynamic ranges (e.g., bandwidth, latency, host capability) with incessant and rapid growth. Physical testbeds allow real-time and high-fidelity experiments on real networks and devices, but the scale and flexibility are limited as it is often too expensive or even infeasible to construct the testing scenarios. Network simulators, on the other hand, provide great flexibility and scalability, but at the cost of fidelity due to the simplification and abstraction of the models. A compelling approach that balances scalability and fidelity is the network emulator that allows direct code execution in virtual machines or containers running on a physical machine.

Authors' addresses: Gong Chen, gchen31@hawk.iit.edu, Illinois Institute of Technology, Chicago, IL, USA, 60616; Zheng Hu, zhenghu@uark.edu, University of Arkansas, Fayetteville, AR, USA, 72701; Yanfeng Qu, yqu@uark.edu, University of Arkansas, Fayetteville, AR, USA, 72701; Dong Jin, dongjin@uark.edu, University of Arkansas, Fayetteville, AR, USA, 72701.

ACM 1049-3301/2023/12-ART
https://doi.org/10.1145/3635307

Network emulators provide rapid prototyping of network applications by leveraging virtualization techniques, such as Xen [9], OpenVZ [1], and Linux Containers (LXC) [18]. While executing binaries instead of abstract models ensures functional fidelity, a network emulator may still face temporal inaccuracy. This is because containers are multiplexed on a single physical machine, and each container uses the same system clock of the underlying machine. Additionally, the execution time and order of the containers are non-deterministic during an experiment as the scheduling is controlled by the host machine's operating system. Therefore, each container's perception of time reflects the serialization of execution on the host machine, but not the execution of their tasks.

To address the temporal fidelity issue, people develop virtual time systems for virtual machines (VM) and containers used in a network emulation experiment [7, 8, 11, 12, 14, 17, 19, 20, 34, 36]. Each virtual node has an independent virtual clock that only advances when the node is in the execution or waiting state. Unchained from the system clock, the nodes can perceive their own virtual time as running independently and concurrently on different physical machines. TimeKeeper is a container-based virtual time system that follows the aforementioned principle [20]. Each container has an independent virtual clock and enables processes that run inside it a notion of virtual time. It circumvents the scheduling of the host machine's operating system and provides a synchronization mechanism to control each container's execution order and duration. Therefore, each container's clock can advance at the same rate. TimeKeeper ensures that the perceived virtual time only advances during the process execution on the CPU. However, the elapsed time on processes should consist of the time elapsed during the execution burst on the CPU as well as the time waiting for non-CPU tasks to complete, such as disk I/O, network I/O, and GPU computation. For simplicity, we use the term I/O to refer to the tasks not running on the CPU, including disk I/O, network I/O, and GPU tasks in this paper. The absence of precise control of I/O time leads to errors in virtual time advancement.

To tackle this issue, we design and implement a virtual time system, named VT-IO , that integrates precise time measurement when a container yields the processor for I/O jobs to complete. However, in large-scale network emulation settings, temporal fidelity issues can still arise even with the use of VT-IO, as a result of the competition for I/O resources among numerous processes, especially when non-CPU resources are heavily utilized. To address this challenge, we extend VT-IO with a time calibration capability in order to mitigate the temporal errors caused by resource contention during concurrent I/O operations. VT-IO effectively controls the execution and time synchronization of containers for both CPU and I/O activities. By making modifications to the Linux kernel, VT-IO enables the tracking of each process's I/O states. At the start of each cycle, VT-IO checks if there are any unattended I/O operations and compensates for the elapsed time during these operations by adjusting the virtual clock of the corresponding process. To assist with the compensation procedure, we introduce a newly developed module called the Dynamic Load Monitor (DLM). DLM monitors the system state, including I/O usage and GPU load, in real-time using tools such as iotop and nvidia-smi. It then utilizes a machine learning-based calibrator to mitigate the temporal errors introduced by resource contention and accurately calculate the precise virtual time advancement. We evaluate the performance of VT-IO with DLM and compare it with our previous work [11] in terms of accuracy in scenarios involving I/O contention. The results demonstrate that the new system significantly reduces the error in I/O time to 0.074 seconds, compared to the original VT-IO with an error of up to 7.889 seconds. These evaluations were conducted in scenarios where 32 containers simultaneously perform intensive I/O jobs, highlighting the effectiveness of our approach in mitigating temporal errors caused by resource contention.

We also demonstrate the advantage of VT-IO in terms of accuracy during I/O operations, synchronization overhead, and system scalability. We compare the results with measurements from a physical testbed, TimeKeeper, and VT-IO. VT-IO reduces the error of I/O time measurement to 3.6%, while the error is up to 87.31% in TimeKeeper. VT-IO is a lightweight modification of Linux Kernel and introduces limited synchronization overhead similar to TimeKeeper. For instance, the synchronization overhead of 512 containers on 64 CPUs is less than 0.498 ms per cycle which is around 0.091% of the overall execution time. The execution time of the same experiment

linearly decreases as the number of containers grows. It is worth noting that our main focus in this paper is centered around the analysis and quantification of blocking operations, specifically disk I/O, network I/O, and GPU operations.

The remainder of the paper is organized as follows. Section 2 compares the existing virtual time systems for network emulation. Section 3 models and analyzes the temporal error of a virtual time system during I/O operations. Section 4 presents the architecture of VT-IO, a virtual time system that integrates precise I/O time management. Section 5 describes the model and architecture design of Dynamic Load Monitor with performance evaluation. Section 6 illustrates the implementation details of VT-IO, including synchronization and I/O time compensation. Section 7 evaluates VT-IO in terms of accuracy, synchronization overhead, and scalability. Section 8 concludes the paper with future work.

2 RELATED WORK ON VIRTUAL TIME SYSTEMS FOR NETWORK EMULATION

Virtualization technologies, such as Xen [9], OpenVZ [1], and Linux Containers (LXC) [18], provide isolated execution environments (e.g., file system, network interfaces, process tree, etc.) for experimenters to execute raw network application code on a physical machine. Such testbeds can raise the issue of temporal fidelity due to the increased computational cost. When the resources on the physical machine, such as CPU, memory, network bandwidth, and disk I/O, are insufficient to serve all the processes, certain processes have to yield and wait for resources while the system clock continues to advance. Additionally, the resource allocation controlled by the host machine does not guarantee fair dispatch for each process.

One effective approach to addressing this issue is to develop and integrate a virtual time system into the network emulation [7, 8, 11, 12, 14–17, 19, 20, 33, 34, 36]. The fundamental idea is to decouple emulated processes from the system clock and assign each process an independent virtual clock that only advances during the process's execution. We compare the existing virtual time systems for network emulation and summarize the results in Table 1.

Table 1. Comparison of Virtual Time System for Network Emulation

System	Virtualization Technique	TDF	per-VM TDF	Adaptive TDF	OS Scheduling Modification	Simulator Integration	Distributed Memory	CPU-Instruction	I/O Compensation
DieCast[14, 15]	Xen, Paravirtualization	✓							
SVEET! [12]	Xen, Paravirtualization	✓				PRIME [21]			
VT-OpenVZ [19, 36]	OpenVZ container	✓		✓	✓	S3F [25]			
TimeKeeper [20]	Linux container	✓	✓		✓	NS-3 [27]			
VT-Mininet[34] DSSnet[16] Minichain[33]	Linux container	✓	✓		✓	S3F[25], OpenDSS[2]			
Distributed VT[17]	Linux container	✓	✓		✓	OpenDSS[2]	✓		
Kronos [7, 8]	Linux container	✓	✓		✓	S3F [25]		✓	
VT-IO [11]	Linux container	✓	✓		✓				✓

One of the pioneering works to study virtual time in network emulation is DieCast [14, 15], which modifies the Xen hypervisor to make the time in a virtual machine (VM) advance more slowly than real-time, and therefore, to scale the resource on the host machine to match the behavior of target applications. DieCast introduces a key variable, *time dilation factor (TDF)*, which is the ratio between the rate at which time passes in the physical world (wall clock time) to the VMs' perception of time (virtual time) [15]. A TDF of ten indicates that the time on the host machine advances ten seconds while the virtual time in the VM only advances one second. As a result, the resources on the host (e.g. CPU, network I/O, and disk I/O) appear to be ten times more powerful. For instance, a 100 MHz CPU is scaled to 1 GHz from the emulated host's viewpoint. Based on this idea, SVEET! [12], a TCP performance evaluation testbed, was proposed to evaluate TCP applications in a high-fidelity setting.

However, DieCast and SVEET! use Xen, which pre-allocates physical resources, such as CPU, and network I/O, to VMs. Each VM has an independent stack of the OS kernel, which could easily use up the physical resources on the host machine and thus affect the scalability of the emulation experiments. Additionally, the execution order and execution burst length on each VM is controlled by the hypervisor, not the emulator. As a result, the virtual time of each VM may not be synchronized, especially on a fine-grained time scale.

To enable high-fidelity and scalable emulation, VT-OpenVZ [36] presents a virtual time system for OpenVZ [1], an OS-level virtualization technology. OpenVZ allows multiple lightweight virtual environments (VE) sharing the same OS kernel. Each VE is assigned an independent virtual clock, and the modified scheduler determines the execution burst length and order for each VE. Therefore, all virtual clocks advance in a synchronization window to improve the temporal fidelity and ensure the causality of the emulated events. Furthermore, other than scaling up the computational resources by slowing down the time (e.g., DieCast), VT-OpenVZ can also speed up the virtual time advancement and thus accelerate the experiments when the workload is light.

To further improve the scalability and flexibility, Lamps et al. proposed a Linux container [18] based virtual time system called TimeKeeper [20]. According to the experimental results, TimeKeeper is highly scalable and is capable of managing 100X more virtual nodes than VT-OpenVZ on the same physical host. TimeKeeper also provides interfaces that enhance the flexibility of the system. For instance, TimeKeeper supports dynamic TDF change and synchronization of LXCs' virtual time, even if the containers have different TDFs [20]. Therefore, users can easily control the speed of their experiments.

Due to the simplicity and precise virtual time advancement, multiple testbeds were built on top of TimeKeeper to evaluate various applications. VT-Mininet integrates a virtual time system into Mininet [3], a software-defined networking (SDN) emulator. Inspired by TimeKeeper, VT-Mininet modifies the Linux kernel to enable TDF, virtual time, and synchronization for each container, which significantly improves the scalability and fidelity of Mininet. DSSnet[16] integrates VT-Mininet with OpenDSS [2], a power distribution system simulator, which expands the influence of virtual time from the cyber-domain to the cyber-physical domain. Furthermore, a distributed virtual time system [17], is developed on top of DSSnet, which enables virtual time synchronization across multiple embedded Linux devices and unchains the bottleneck of emulation tied to a single physical host. Mininchain [33], demonstrates the usability of virtual time supported emulation in a real-world application. It directly executes the unmodified Ethereum code [32] in the containers and utilizes TDF to speed up blockchain emulation experiments.

While most existing approaches derive the virtual time based on the system clock, Kronos [7, 8] is a new method to advance virtual time based on CPU instructions. Instead of controlling how long processes can be executed on the CPU using hrtimer [13] and signaling techniques [10], Kronos controls a process's virtual time based on the binary instruction counts measured by ptrace [6] and perf [5]. Compared with the timer-based virtual time systems, Kronos enhances the scalability and temporal fidelity of the emulation system. Kronos also enables the precise quantification of the computational power of containers. Each container is specified with a variable, called relative CPU speed, which indicates the number of instructions processes can complete per second. For instance, a relative CPU speed of one million indicates that the process advances for one second if one million instructions have been executed. Kronos leverages the relative CPU speed to translate the instruction counts to

virtual time. Therefore, Kronos improves the repeatability of the emulation experiments because the virtual time advancement is completely decoupled from the host machine and is only dependent on the application and the configuration of emulation (e.g., relative CPU speed).

Virtual time system, such as TimeKeeper and Kronos, ensures precise virtual time advancement when the application code is executed on CPUs. However, the effective elapsed time on processes should consist of the time elapsed during the execution burst on the CPU as well as the time waiting for I/O tasks to complete. Lack of I/O time management would largely impact the temporal fidelity of emulation. In this work, we propose VT-IO , a virtual time system that integrates precise virtual time for both CPU and I/O intensive tasks. We also develop a module called Dynamic Load Monitor that enables precise I/O advancement even under serve resource contention scenarios. The detailed analysis and system design will be presented in Sections 4 and 5.

3 ERROR ANALYSIS OF VIRTUAL TIME SYSTEM WITH I/O INTENSIVE TASKS

Existing works of virtual time systems in network emulation fall into the following two categories: 1) timer-based approaches [12, 14, 17, 19, 20, 34, 36] that rely on the operating system's clock to control the execution of emulated processes and 2) instruction-count based approaches [7, 8] that map the advancement of virtual time to the number of assembly instructions executed by emulated processes. Both designs enable each Linux container and the emulation processes to advance their clocks independently from the system clock during the execution. This section explores the limitations of the existing virtual time systems during intensive I/O operations. In particular, the virtual time is not correctly computed when a Linux container is waiting for I/O-intensive jobs to complete, such as disk I/O, network I/O, or GPU computation. We use TimeKeeper [20] as a demonstrative case since many recent works [7, 16, 33, 34], including this work, are inspired by the design of TimeKeeper. We mathematically model the temporal error caused by I/O operations to illustrate the necessity of integrating precise I/O time control into the virtual time system.

3.1 Virtual Time Advancement in TimeKeeper

TimeKeeper is a lightweight virtual time system for Linux container (LXC) [18] based network emulation. While full virtualization (e.g., VMWare [29], VirtualBox [30]) and paravirtualization (e.g., Xen [9]) require a separate kernel for each virtual machine instance, LXC takes a lightweight approach by allowing multiple Linux instances running on a shared kernel. TimeKeeper assigns an individual virtual clock to an LXC. The same virtual clock is shared by all the processes and their child processes inside the LXC. Each virtual clock is associated with a time dilation factor (TDF) [15], which is defined as the ratio between the rate at which wall-clock time has passed to the emulated host's perception of time. For instance, a TDF of 2 means that processes in a time-dilated LXC perceive the time advancement as one second for every two seconds of wall-clock time. In other words, time is passed two times slower in the LXC than in the real world. A TDF of 0.5, on the other hand, indicates that the virtual time in the container advances two times faster than the real-time.

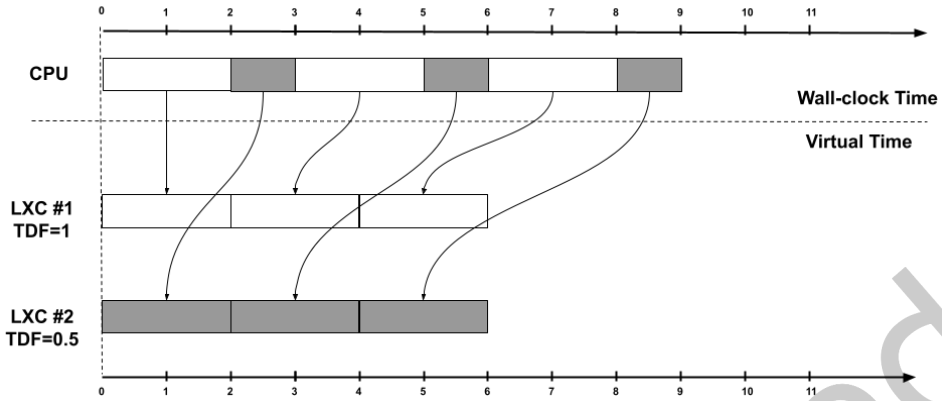


Fig. 1. Virtual time advancement for CPU intensive processes

TimeKeeper proposed a barrier-based conservative synchronization mechanism [24] to synchronize the virtual time among the containers. The advancement of an emulation experiment is divided into many small execution cycles. Containers with different TDFs get their proportional execution time in wall-clock time, while the virtual clock of each container advances the same amount during each cycle. In other words, TimeKeeper uses a barrier to control the virtual time of each container to advance at the same rate. Containers are executed cycle by cycle, and their virtual time is adjusted and synchronized at the barrier between the cycles. Each container is assigned a specific execution time when a cycle starts. TDF and quanta determine the execution time. Quanta is a user-defined parameter denoting the time granularity of one cycle. Theoretically, a smaller quanta offers better temporal accuracy despite the additional overhead due to frequent synchronizations.

TimeKeeper enables the parallel execution of containers with multiple CPUs. At the beginning of synchronization, each container is designated to a CPU with a high priority, which guarantees that the container is executed on the specific CPU and will not get preempted by another process. Multiple containers can be assigned to one CPU. For example, Figure 1 illustrates how two containers with different TDFs are scheduled on one CPU and how their virtual time is synchronized. *LXC* #1 has a TDF of 1, so its virtual clock advances at the same speed as the wall clock. The TDF of *LXC* #2 is 0.5, which means its clock advances two times faster than the wall clock. *LXC* #1 and *LXC* #2 are scheduled to be executed on the CPU in turns. In each cycle, the execution time of *LXC* #1 is twice longer than the one of *LXC* #2, so both virtual clocks can advance at the same rate.

3.2 Demonstration of TimeKeeper Imprecision with I/O Intensive Tasks

TimeKeeper and other timer-based virtual time systems [7, 16, 33, 34] well manage the virtual time advancement during CPU burst cycles. However, the virtual time imprecision occurs when the CPU waits for I/O operations (e.g., disk or network) or GPU computation to complete. Figure 2 demonstrates how a temporal error is formed in a container (*LXC* #1) after the execution of an I/O operation. We simplify the scenario by setting the TDF of both containers to 1. *LXC* #1 initiates an I/O operation at 3 in virtual time (i.e., at 5 in wall-clock time). The I/O operation takes 5 seconds to complete at 10 in wall clock time, while *LXC* #1 finishes the third cycle. However, the elapsed virtual time in *LXC* #1's perception is only 3 seconds instead of 5 seconds. Therefore, this I/O appears to have finished at 6 in virtual time. The 2-second temporal error in *LXC* #1 is due to the lack of control of virtual time during I/O operations in TimeKeeper. The I/O operations continue their execution even if the corresponding container is paused for synchronization. The container cannot observe the elapsed time for the ongoing I/O once

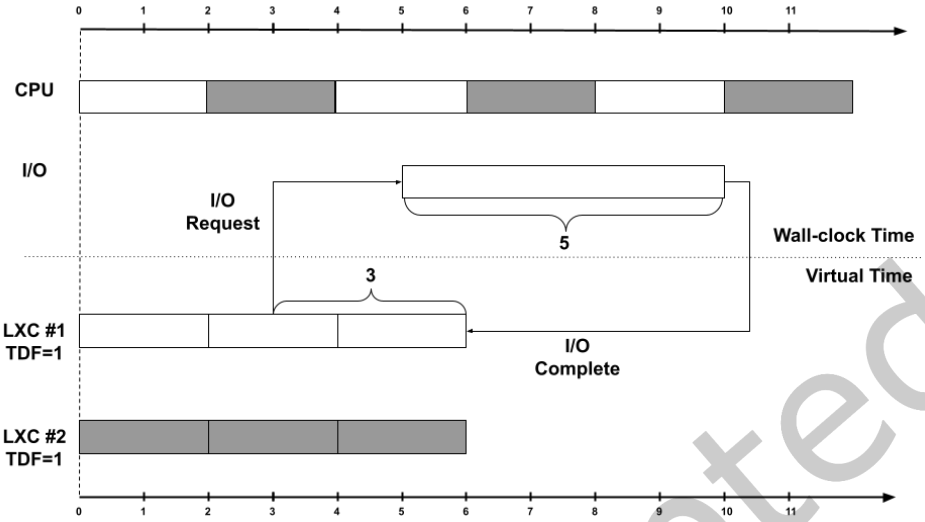


Fig. 2. Virtual time advancement with I/O operations in Timekeeper. In this example, I/O time is 5 seconds and thus *LXC* #1 should advance its virtual time for 5 seconds instead of 3 seconds (TDF = 1), i.e., no integration of I/O time in TimeKeeper causes a temporal error of 2 seconds.

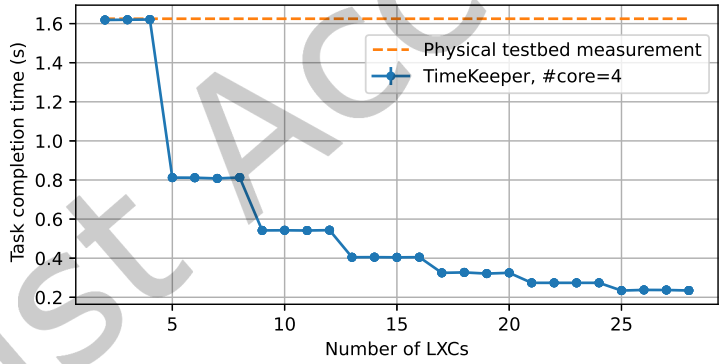


Fig. 3. Demonstration of temporal error: measurement of the GPU task completion time in TimeKeeper

it is paused. As shown in Figure 2, from time 6 to 8, it is *LXC* #2's turn to be executed on the CPU while *LXC* #1 is paused. As a result, *LXC* #1's clock does not advance for 2 seconds while the I/O operation is executing. The missing I/O time is the temporal error we need to address in this paper. If we set up more containers to share the CPU in the experiment, we will have a larger temporal error with an even lesser I/O time for *LXC* #1.

We empirically demonstrate the timing imprecision and show the experimental result in Figure 3. Only one container conducts I/O tasks in the experiment, while the others run CPU-intensive applications. For demonstration purposes, the I/O task in this experiment is matrix addition operations on GPU. The other types of tasks, such as disk I/O and network I/O, are discussed in Section 7. Figure 3 depicts the I/O completion time by

varying the number of containers. Task completion time is the virtual time elapsed during an I/O operation. The dashed orange line is the measurement on a physical testbed as the ground truth. The difference between the two lines is the temporal error. The error increases as the number of containers grows. We discovered an interesting pattern: the completion time drops when the number of containers increases by four. Four is also the number of CPUs used in the experiment. The following section mathematically models the error to explain the observed pattern.

3.3 Error Modeling

Suppose there are n containers denoted as l_i, where $0 < i \leq n$, in an emulation experiment running on a machine with m available CPUs, denoted as c_j, where $0 < j \leq m$. Containers are scheduled on the CPUs in a round-robin fashion. Multiple containers can be assigned to the same CPU if $n > m$. Each container l_i is associated with a time dilated factor (TDF), tdf_i. The container with the largest TDF is called the leader, and its TDF is denoted as tdf'. TimeKeeper uses conservative synchronization to keep all the containers' virtual time advancing at the same pace. At the beginning of each execution cycle, the system calculates the execution time in wall clock, δ_i, for each container l_i based on tdf_i and a quanta ϵ. Quanta is a predefined constant representing the granularity of time synchronization in emulation. δ_i is calculated with the following equation.

$$\delta_i = \frac{\epsilon \times tdf_i}{tdf'} \tag{1}$$

Equation 1 ensures the containers' virtual clocks are synchronized at each cycle. The execution time, α_j, of all containers assigned to each c_j in one cycle is calculated as follows. The function F yields all containers associated with a CPU.

$$\alpha_j = \sum_{\forall i \in F(c_j)} \delta_i \tag{2}$$

The total wall clock time elapsed in one cycle is

$$T = \max\{\alpha_j | 0 < j \leq m\} + \mu \tag{3}$$

where μ is the synchronization overhead time between cycles.

Assume a container l_i initiates a request for an I/O operation, and the time to complete the I/O is Δt in wall-clock time. Equivalently, it takes $\Delta t / tdf_i$ for l_i to complete the task in virtual time. If $\Delta t / tdf_i$ is small enough for the I/O to complete within δ_i, the resulting virtual time is accurate, and otherwise, the error occurs. Because l_i is still waiting for I/O even though l_i is paused for synchronization (i.e., the virtual clock of l_i is frozen). The I/O operation time can be modeled as follows.

$$t \approx \frac{\epsilon \times \Delta t}{tdf' \times T} \tag{4}$$

Given a constant and unified TDF, we can further simplify Equation 4 and deduce the following relation to explain the observation in Figure 3.

$$t \propto \frac{\Delta t}{\lceil n/m \rceil} \tag{5}$$

To further validate Equation 5, we conducted another experiment in TimeKeeper running on two CPUs. TDFs are set to 1 for all containers. One container initiates a disk I/O while the others run a CPU-intensive application. Figure 4 depicts the elapsed virtual time during the disk write operation, and the result well matches the relation shown in Equation 5.

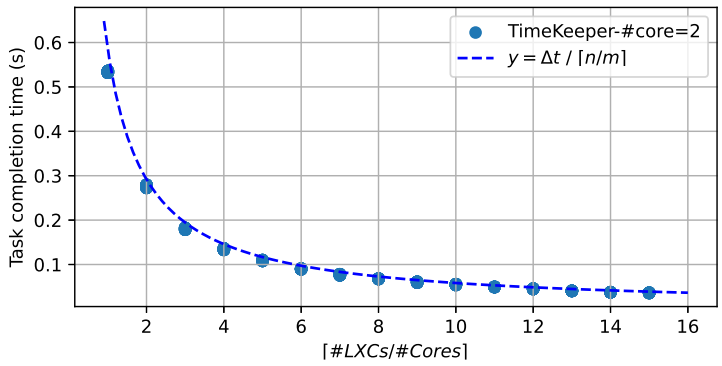


Fig. 4. Measurement of disk write operation completion time in TimeKeeper. n is the number of containers. m is the number of CPUs. Δt is a measurement of time elapsed during the write operation in the wall clock. TDF of all containers in this scenario is 1.

3.4 I/O Temporal Error Mitigation

Containers are executed in cycles for virtual time synchronization. T is the total execution time in each cycle (see Equation 3). Each container is executed on a CPU for δ_i time (see Equation 1). δ_i is usually less than T since multiple containers share the CPU. Therefore, the containers are paused for $T - \delta_i$ in each cycle, during which the containers yield their CPU, and the virtual clock is stopped. However, if an I/O operation cannot be completed in δ_i, the remaining I/O time will not be counted while the virtual clock is stopped. As a result, the recorded I/O time seen by the container is shorter than the actual time. Such error in each cycle is upper bounded by $(T - \delta_i)/tdf_i$ and the error accumulates over cycles. For instance, if an I/O takes k cycles to finish, the total error is $k \times (T - \delta_i)/tdf_i$. Since $k \simeq \Delta t/T$, the total error is modeled as follows.

$$E \approx \frac{\Delta t}{T} \times (T - \delta_i) \tag{6}$$

To mitigate the temporal error during I/O, we designed a new module called I/O Time Compensator. The compensator checks each container's state at the end of its execution cycle. If an ongoing I/O operation is detected, an active I/O flag is set for the container. At the beginning of the next execution cycle, the compensator calculates the I/O time and adds it back to the virtual clock of the container whose active I/O flag is true. How to precisely calculate and compensate for I/O time is discussed in detail in Section 4.2 and Section 6.1.

Synchronization Controller is designed to synchronize virtual time among the containers (see Section 4.2 for details). In particular, it determines the execution time of each container and the execution order at the beginning of each cycle. The execution time is defined in Equation 1. However, once the I/O time compensator is triggered, certain containers' virtual clocks may be ahead of the others, affecting the system's temporal fidelity and emulation causality. Therefore, we propose an adaptive scheduler doing dynamic execution time calculations to synchronize the virtual time gradually. The details of the scheduler are described in Section 6.2.

4 DESIGN ARCHITECTURE

We design a virtual time system named VT-IO that integrates precise virtual time measurement when a container yields the processor and waits for jobs such as disk I/O, network I/O, or GPU computation to complete. VT-IO has two layers: 1) Container-based Middleware that offers each container a perception of virtual time, and 2)

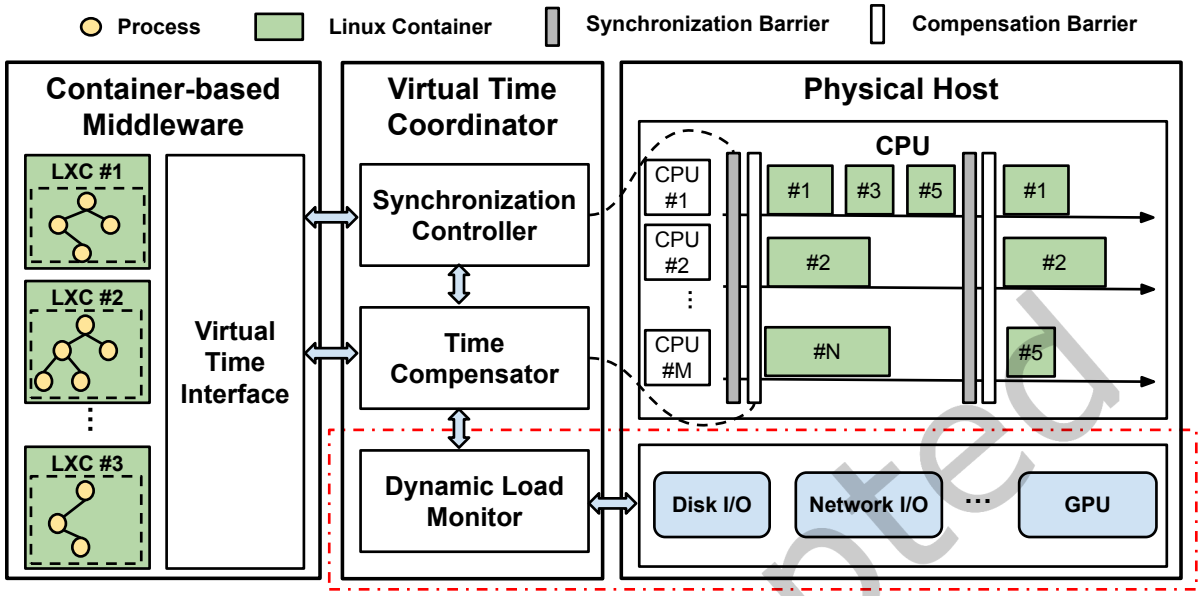


Fig. 5. Architecture design of VT-IO with Dynamic Load Monitor

Virtual Time Coordinator that controls the execution and virtual time synchronization of containers on CPU and I/O activities. The architecture design of VT-IO is depicted in Figure 5.

4.1 Container-based Middleware

The container-based middleware consists of lightweight Linux containers and a unified virtual time interface. A container in VT-IO is a predefined template that offers an OS-level virtualization environment and encapsulates virtual-time-related variables, such as TDF and execution time, for processes running inside the container. New container instances are initiated when a network emulation experiment starts, and each container emulates an individual physical host. Each container maintains its own virtual clock, and time dilates the processes and their child processes running inside the container. No application code modification is necessary to enable virtual time for the processes.

The virtual time interface intercepts and handles the time-related functions, such as `gettimeofday` and `nanosleep`. If a process inside a container invokes a time-related system call, the default system call is circumvented, and the call is redirected to a modified function whose return value is based on the container's virtual clock.

The middleware and virtual time interface collaboratively offer the emulation processes a notion of virtual time. Their implementation details are discussed in Section 6.

4.2 Virtual Time Coordinator

It is challenging to keep all the containers advancing the virtual time with the same amount each cycle. By default, the operating system schedules the execution of each process. The order and the execution time are both non-deterministic. It is impossible to distribute the CPU time among all the containers evenly. Therefore, the processes running in the container have to face temporal fidelity issues, especially when the resources (e.g.,

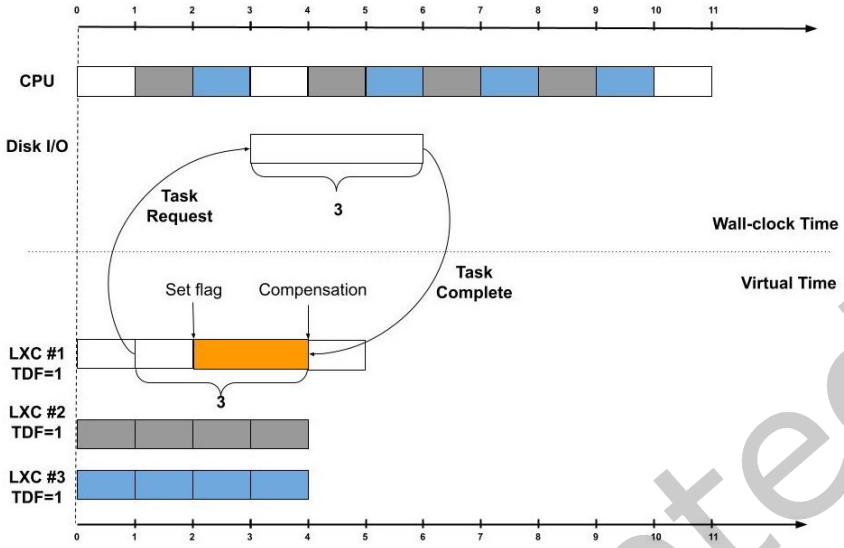


Fig. 6. Virtual time advancement with I/O operations in VT-IO

available CPUs, network bandwidth) on the physical machine are insufficient to support a large number of containers [34].

To address this problem, researchers have explored the virtual machine scheduling mechanism [31, 36], based on which existing works have proposed various designs for Xen [12, 14, 15] and Linux Container [7, 20, 36]. In this work, we design the Virtual Time Coordinator to handle the synchronization of containers for CPU execution and I/O operation. Virtual Time Coordinator contains three modules: Synchronization Controller, Time Compensator, and Dynamic Load Monitor, as shown in Figure 5.

Synchronization Controller is invoked at the beginning of each execution cycle. The 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 controller then assigns each container to a designated CPU and schedules the execution of processes inside the container in a round-robin style. A hill-climbing algorithm is used for scheduling. For each container, the scheduling algorithm selects a CPU core that has minimal workload at the moment and assigns the container to it. It yields an overhead bounded by $O(m \times n)$, where m is the number of designated CPU cores and n is the number of containers. While exploring an efficient scheduling algorithm for emulation containers is not a focus of this work, it is worth mentioning that the current scheduling algorithm is fast but may not always yield the optimal solution. One approach is to use a fine-tuned scheduling algorithm [35], but it inevitably introduces more overhead due to the increasing time and space complexity of the algorithm. We will leave the design and analysis of efficient container scheduling algorithms as future works.

Time Compensator is invoked at the end of each execution cycle to compensate for the missing virtual time during I/O operations. The I/O operations keep running even though the associated container yields the CPU. The virtual time is stopped at the end of its execution, but the container may not complete the I/O task by the end of the container's assigned execution time. Temporal errors arise as the remaining I/O time should have been included. Time Compensator is designed to mitigate this temporal error. It checks the status of every LXC at the end of each execution cycle and determines whether an I/O task is ongoing. If true, the missing time measurement for this task will be compensated back to the corresponding LXC at the next cycle.

Figure 6 demonstrates an example of how the Time Compensator adjusts the container's virtual clock to include the I/O time. There are three containers, all with a TDF of 1 in the scenario. *LXC* #1 initiates an I/O operation at time 1, which is the beginning of the second cycle. This operation takes 3 units in wall-clock time to finish. At time 2, *LXC* #1 finishes its execution time for the second cycle, but the I/O is not yet completed. Therefore, *LXC* #1 is blocked. The compensator detects this blocking state and sets a flag and a timestamp t_1 to *LXC*#1 (i.e., 4 in wall-clock time). After that, the I/O operation continues while *LXC* #2 and *LXC* #3 run for their second cycle. The I/O completes at the end of the second cycle t_2 (i.e., 6 in wall-clock time). Before the third cycle starts, the compensator checks the flag of containers and updates *LXC* #1 's virtual clock by adding $(t_2 - t_1)/tdf_1$, i.e., (6-4)/1 = 2. Therefore, the virtual time of *LXC* #1 elapsed during the I/O operation is compensated. Since the clock of *LXC* #1 is ahead of the other two containers, *LXC* #1 will yield the execution cycles until the other containers catch up.

Dynamic Load Monitor enhances the temporal fidelity of VT-IO, especially under resource contention scenarios. Although the Synchronization Controller and Time Compensator enable precise virtual time measurement, VT-IO still yields temporal error due to resource contention for I/O-intensive tasks. To tackle this problem, we designed a new module, Dynamic Load Monitor, to calibrate the virtual time advancement based on the load of the host machine.

The workflow depicted in Figure 7 illustrates the operation of the Dynamic Load Monitor. This component works in conjunction with the Time Compensator to update the virtual clock of a container that has an incomplete I/O intensive task. It also collaborates with the Synchronization Controller to adjust the scheduling for the subsequent cycle. This adjustment is necessary because the virtual clock of the updated container may be ahead of the other containers due to the compensation process. At the start of each cycle, the VT-IO examines whether an LXC (Linux Container) had an unfinished I/O or GPU task in the previous cycle. This information is conveyed through the compensation flag, as shown in Figure 7. When the flag is set to False, VT-IO follows the standard path without employing dynamic compensation, indicated by the black arrows. However, if the flag is True, VT-IO activates the dynamic time compensation mechanism, and the execution path is depicted by the red arrows. The Time Compensator performs an accurate calculation of the missing virtual time for the unfinished task. Subsequently, the Dynamic Load Monitor assesses the system load and employs machine learning models to adjust the time measurement based on the load. This adjustment is used to update the virtual clock of the corresponding LXC. Finally, the Synchronization Controller schedules the execution of each LXC to ensure that their virtual time remains synchronized by the end of the cycle. Further details regarding the design, implementation, and evaluation of the Dynamic Load Monitor are discussed in the subsequent section.

5 DYNAMIC LOAD MONITOR

In the original design of VT-IO [11], the precise I/O time is enabled at the OS level with the notion of virtual time. However, this measurement can vary dramatically during resource contention, such as disk I/O, GPU, and network I/O, among multiple containers. For example, suppose ten thousand containers simultaneously issue the disk write operations. The measured completion time for one write operation can take much longer than the scenario when only one container performs one write operation, and thus raises a temporal fidelity issue of the virtual time system.

We empirically demonstrate the timing imprecision under I/O resource contention and show the experimental results in Figure 8. In this experiment, multiple containers simultaneously write 100 MB data to the disk and hit the disk I/O bottleneck. The task completion time is the virtual time elapsed during a disk write operation. The boxplot shows the distribution of the task completion time with different numbers of LXCs. The orange line shows the trend of the median among different LXCs. We observe that the task completion time significantly increases as the number of LXCs grows due to the higher level of I/O resource contention in the physical machine.

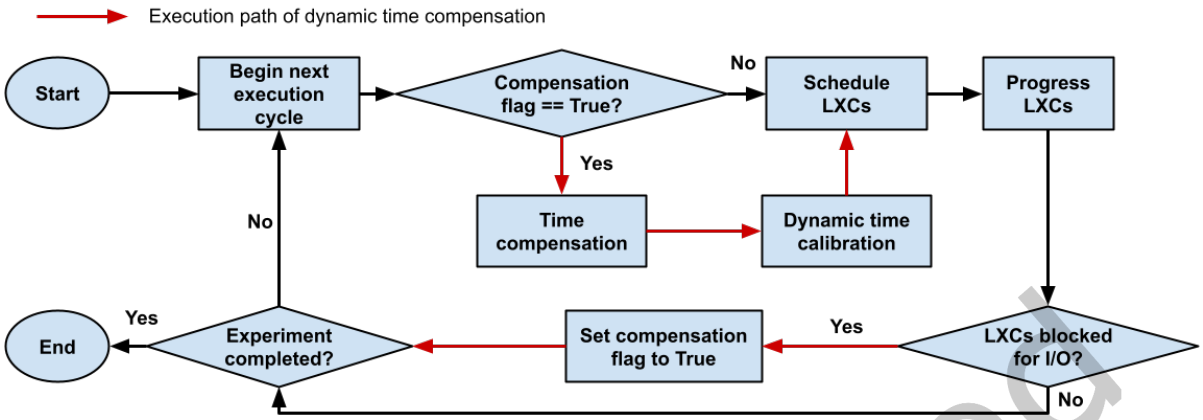


Fig. 7. Workflow of I/O time compensation and dynamic time calibration in an execution cycle

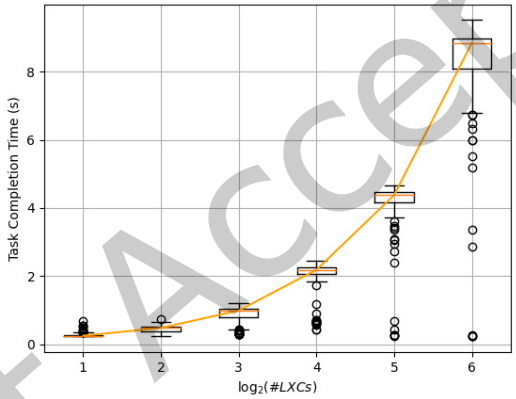


Fig. 8. Motivation example: temporal error of VT-IO under I/O resource contention

5.1 Design and System Integration

Existing virtual time systems mitigate temporal errors caused by CPU resource contention through the precise scheduling and control of the execution bursts of processes. A timer-based virtual time system [16, 20, 33, 34] leverages the Linux Signals (e.g., SIGSTOP and SIGCONT) and high-resolution timer [13] to manage the execution of each process in the containers, while an instruction-based virtual time system, such as Kronos [7] uses ptrace [6] and iperf [5]. However, the execution control of I/O operations is not as easy as CPUs. Limited technologies are available to control the execution of I/O operations. Once a user space process initiates an I/O request (such as sys_read or sys_write), the request will be handled by the OS block layer [23]. The block layer simultaneously mediates and orchestrates the I/O requests from multiple applications to the underlying resources, such as disk I/O. The schedule and dispatch of the requests depend entirely on the I/O scheduling policy and the device driver. A heavy intrusive modification of the I/O stack of the Linux kernel is required to enable controlling and synchronization for IO-intensive tasks. The modification includes a generic block layer, BIO, scheduling of the

request and dispatch queue, and service routine of the device driver. The modification is not only labor-intensive but also challenging since modifying each of the aforementioned modules could lead to OS corruption.

We propose a different approach that involves limited modifications to the kernel and provides a more general solution across multiple operations. Instead of individually considering contention in each layer, we treat them as a collective whole. This allows us to model and abstract the time compensation and calibration mechanism at a higher level, making it more applicable to all the operations discussed in the paper. By taking a higher-level approach, we aim to provide a more practical and efficient solution that reduces the complexity and potential risks associated with deep modifications at each layer. Our goal is to develop a general solution that can effectively address contention-related issues across various types of operations without extensively modifying the kernel for each specific scenario.

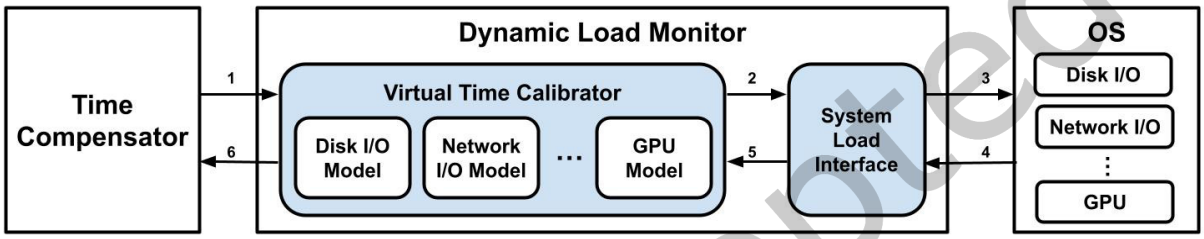


Fig. 9. Dynamic Load Monitor architecture that depicts all steps for I/O adjustment involved in the interaction among Time Compensator, Dynamic Load Monitor, and OS in one round. Each I/O device is associated with a specified pre-trained linear regression model.

We design a new module called Dynamic Load Monitor (DLM) that monitors the host machine's disk I/O, network I/O, and GPU computation activities in real time. The architecture design of the Dynamic Load Monitor is illustrated in Figure 9, which contains two sub-modules: 1) System Load Interface and 2) Virtual Time Calibrator.

System Load Interface provides APIs for querying the status of I/O devices on the host machine, such as the number of processes performing disk writes and the GPU utilization. We use different techniques for different types of operations, e.g., disk I/O API utilizes iotop that monitors the I/O usage information in the Linux Kernel, while nvdia-smi is used to monitor and manage Nvidia GPUs. Our current load monitor only supports the Nvidia GPU series and a device-specific modification is required to support other GPU series, such as AMD.

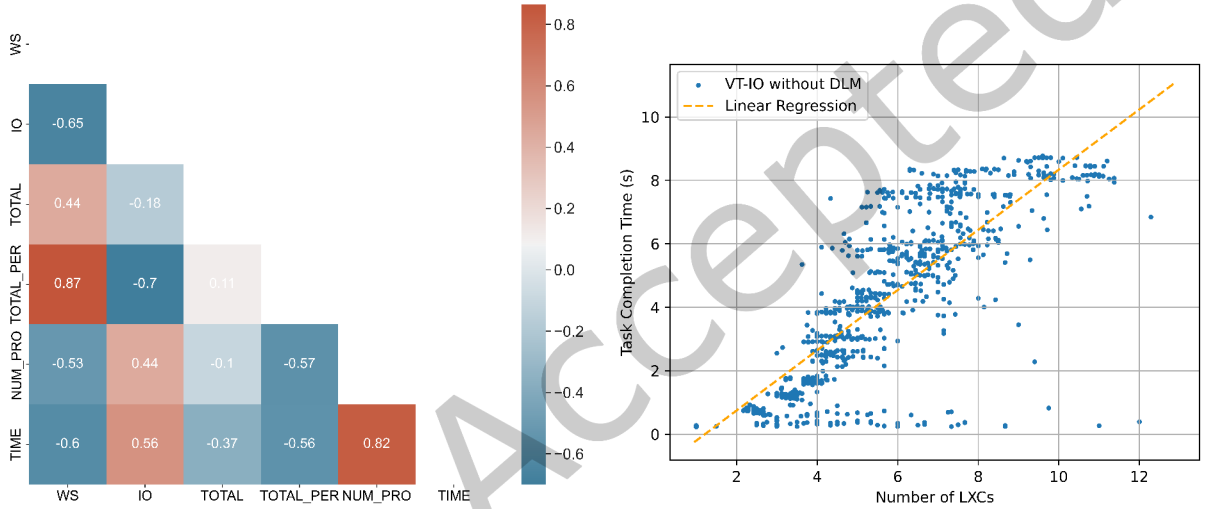
Virtual Time Calibrator is a machine learning based surrogate (Linear Regression) that adjusts the I/O time based on the load on the host machine. It is invoked at the end of each execution cycle once the time compensation is triggered. The calibrator takes inputs from both Time Compensator (e.g., virtual time measurement and type of operation) and System Load Interface (e.g., device status) and outputs an adjusted virtual time. Each type of operation, such as disk write and GPU operation, is associated with a specific model. The selection of models and features also depends on the corresponding I/O type and the details are presented in Section 5.2.

The procedure of dynamic virtual time calibration is hierarchical. Figure 9 demonstrates the cooperation of the Time Compensator, Dynamic Load Monitor, and OS to complete a dynamic virtual time calibration. The request is initiated by Time Compensator and forwarded to Virtual Time Calibrator. The request contains information about the virtual time measurement and the task type. The system Load Interface then invokes APIs for device status queries. The data are fed to the machine learning models in Virtual Time Calibrator. Models then yield an adjusted virtual time and return it to Time Compensator, which finishes the compensation with the calibrated virtual time and continues the next cycle.

5.2 Analysis and Modeling

We analyze the behavior of I/O operations on the host machine and take a machine learning based approach to model the behavior. As shown in Figure 8, the temporal error grows with the increasing number of processes. Other than the number of processes, other factors in the system could also affect the temporal measurement during I/O resource contention. In this work, we select the following five features and analyze the correlation between them and virtual time.

- `WS`: disk write speed (MB/s) for a target process
- `IO`: percentage of time that a process spends on waiting for I/O
- `TOTAL`: total disk write speed (MB/s) for all processes
- `TOTAL_PER`: percentage of disk bandwidth used by a target process
- `NUM_PRO`: number of processes performing I/O operations



(a) (b)

Fig. 10. (a) Heatmap of Pearson correlation coefficient matrix among virtual time and the five selected disk I/O features; (b) correlation between task completion time and number of LXCs that perform disk I/O

We analyze the correlation between the five features and virtual time using the Pearson Correlation Coefficient (PCC), which defines the strength and direction of a linear relationship between two variables [26]. Equation 7 shows the definition of PCC, where x and y are two features of discussion, n is the sample size, and r is the PCC ranging from -1 to 1. The positive or negative sign of r indicates the direction of the correlation, and the value of r implies the strength of the correlation. The closer the value is to -1 or 1, the stronger the correlation is between the two features.

$$r = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2 (y_i - \overline{y})^2}} \tag{7}$$

We conduct a set of disk write experiments to analyze the correlation. The experiments measure the virtual time elapsed for the I/O operations by varying the number of processes. Figure 10a demonstrates the PCC of the five features of disk I/O on the host machine and the measurements of virtual time in VT-IO without DLM.

The relationship between two variables is often considered strong when the PCC value is larger than 0.7 [28]. We observe in Figure 10a that PCC between the virtual time and the number of processes has the largest value among all the pairs and the value 0.82 indicates a strong correlation. The disk write speed and the percentage of bandwidth usage also have a strong relationship with 0.87 PCC. However, the main objective of our model is to minimize the temporal error of the virtual time. Therefore, we focus on the relationship between virtual time and system loads. Other than the number of processes, the PCC values of the other features are less than or equal to 0.6, which is considered a moderate or weak correlation. Therefore, we use the number of processes as the factor to represent the current system load and develop a linear regression model to quantify its relation with virtual time. Figure 10b shows the task completion time (in virtual time) of the regression model by varying the number of processes. A large number of processes indicate a heavy workload. We observe that the virtual time increases linearly with the system load. The mathematical representation of the linear regression model is defined in Equation 8, where t_c is the expected I/O time with the calibration, t_m is the measured I/O in virtual time, μ is the number of processes which represents current system load, and w and b are the coefficients of the model.

$$t_c = w \times \frac{t_m}{\mu} + b \tag{8}$$

5.3 Performance Evaluation

Dynamic Load Monitor enhances the ability of VT-IO to precisely advance the virtual clock during I/O operations, especially when the I/O resources on host machines are limited. Two experiments were conducted to compare the temporal fidelity of VT-IO, with and without the assistance of Dynamic Load Monitor. We measured the virtual time elapsed for each scenario: (a) time of direct writing 100 MB data to disk and (b) time of performing 1,000 iterations of matrix additions on GPU. To demonstrate the impact of the I/O resource competition and the effectiveness of Dynamic Load Monitor, all the containers simultaneously performed the I/O operations, which significantly overwhelmed the I/O resources. We plot the physical testbed measurements as the ground truth as well as the results of VT-IO with and without Dynamic Load Monitor in Figure 11a and Figure 11b.

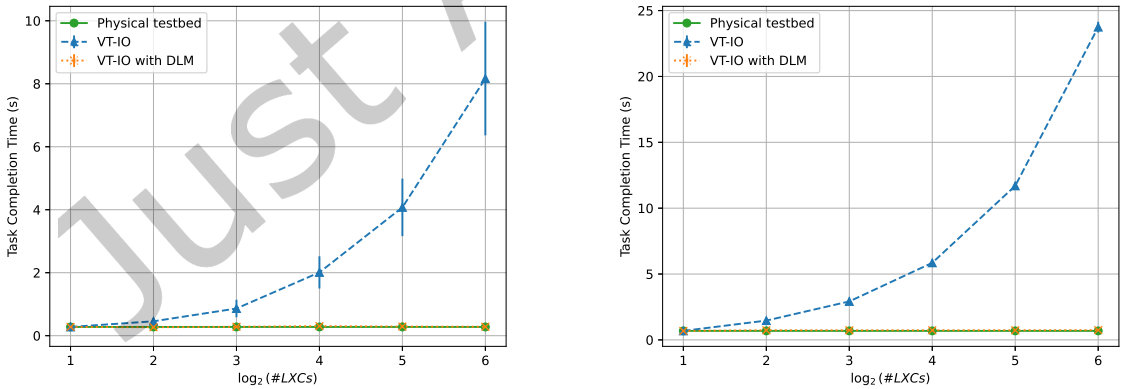


Fig. 11. Measurement of virtual time elapsed during (a) disk I/O operation (direct write of 100 MB data) and (b) GPU operation (1,000 iterations of matrix additions) on a physical testbed (i.e., ground-truth measurement), VT-IO and VT-IO with Dynamic Load Monitor (DLM)

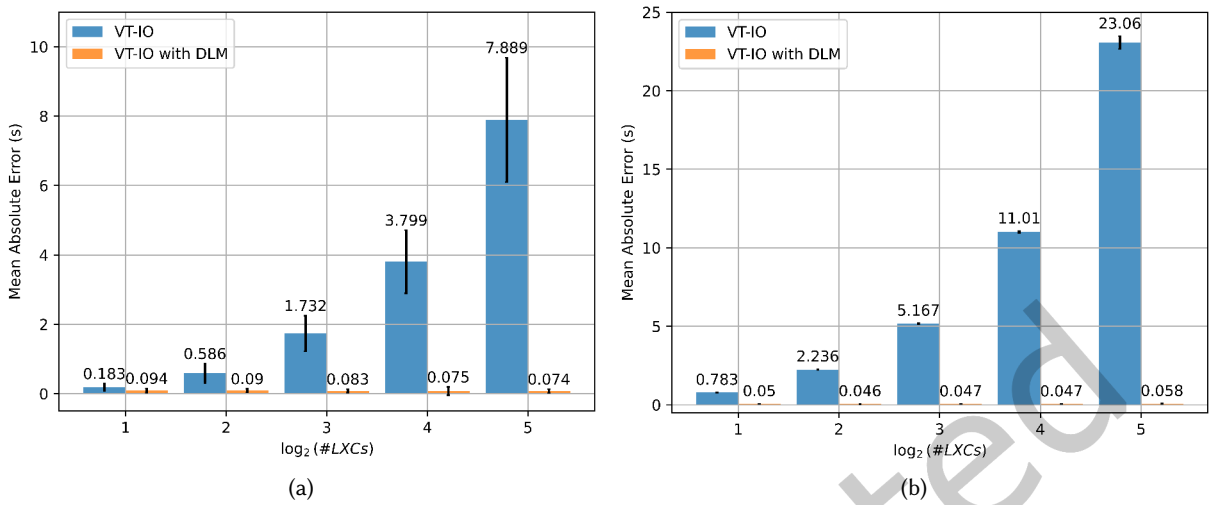


Fig. 12. Mean absolute error (MAE) of virtual time elapsed during (a) disk I/O operation (direct write of 100 MB data) and (b) GPU operation (1,000 iterations of matrix additions) on VT-IO and VT-IO with Dynamic Load Monitor (DLM) compared with the ground-truth measurement on a physical testbed

We observe in both disk I/O and GPU scenarios that the task completion time in the original VT-IO increases as the number of contains grows. The root cause is resource contention. As the number of containers grows, more processes compete for the I/O resources (such as disk I/O bandwidth) on the host machine while the available resource is limited. Therefore, the time spent to finish the same amount of work increases as the number of containers grows. On the other hand, VT-IO with DLM matches well with the physical testbed results. It is because DLM is capable of monitoring the I/O load in real time and dynamically calibrating the virtual time to mitigate the temporal error caused by resource contention. In other words, it precisely maintains the correct virtual time even when the I/O resources are heavily overwhelmed.

Figure 12a and Figure 12b further demonstrate the Mean Absolution Error (MAE) of the virtual time compared with the physical testbed. The error of the original VT-IO significantly increases when the number of containers increases, while the error of VT-IO with DLM is very small and stable. VT-IO with DLM outperforms the original VT-IO in all cases. For instance, in the GPU experiment with 32 containers, the error of GPU operations using the original VT-IO is 23.06 seconds, while the error of VT-IO with DML was reduced to 0.058, which is about three orders of magnitude less.

We notice that the disk I/O measurement on the original VT-IO yields much more significant variances compared with the GPU case. It is because the disk is a physical device and its speed (i.e., data transfer rate) depends on many factors, such as the rotational speed of the disks and the data recording density, and those factors may vary over time. GPUs, on the other hand, are electronic devices with stable performance over time.

6 IMPLEMENTATION

We have modified the Linux kernel to implement VT-IO, a virtual time system, based on the architecture design shown in Figure 5. Container-based Middleware intercepts the timing-related system calls in Linux to enable the virtual time of the containers. Similar to the existing container-based virtual systems [7, 20], we modify the data structure `task_struct`, which is a process descriptor containing every relevant information about a process [22].

We develop a lightweight container with a virtual clock by adding 7 new fields to the `task_struct` with 40 bytes in total.

- `isVirtual`: a boolean variable indicating whether a process is a normal process using the system clock or a dilated process using the virtual clock
- `TDF`: time dilation factor of a process
- `virtualTime`: the elapsed virtual time since the most recent virtual clock update
- `timeToUpdate`: the corresponding wall clock time when the virtual clock is updated most recently
- `compensationFlag`: a boolean variable indicating whether a process is blocked for an ongoing I/O operation
- `falseStart`: the amount of virtual time that a process is ahead of the expected time due to I/O compensation
- `ioCompleteTime`: the completion time of an I/O operation in wall-clock time

Virtual Time Interface circumvents the default time-related system calls and redirects them to our modified functions. The function `updateVirtualTime()` in Algorithm 1 defines the virtual clock update procedure. The function `new_gettimeofday()` is an example of system hijacking that distinguishes the time-related system call between normal and dilated processes. As shown in `new_gettimeofday()` of Algorithm 1, the execution path is controlled by the field *isVirtual*. If a process is dilated, its *isVirtual* field is set to 1. Therefore, the function returns a virtual time. Otherwise, the function returns the system time for normal processes.

We use signaling technology, such as SIGSTOP and SIGCONT, and hrtimers [13], and a high-resolution timer provided by the mainstream Linux kernel, to control the process execution in the containers. Algorithms 2 and 3 illustrate the time compensation procedure. The `pause()` and `resume()` functions describe how to pause and resume a process on the CPU and how to update its virtual clock. Different from TimeKeeper [20], we define new fields and functions to enable a container to track the state of its I/O operations and update its virtual clock for those operations. We design a new barrier to check the I/O state when a process is paused. Based on the state, the field `compensationFlag` in the process descriptor is also updated. When the process is ready to resume, we check the `compensationFlag` to determine whether the I/O operation overshoots the expected execution time. If so, we adjust the virtual clock to compensate for the elapsed time during the I/O operation.

6.1 Precise I/O Time Compensation

One straightforward approach to calculating the I/O time is based on the I/O state of a process. We can calculate the total execution time T in each cycle based on Equation 3 and the execution time δ_i of each container based on Equation 1. Since the I/O error only occurs when a container is paused, i.e., $(T - \delta_i)/tdf_i$. Therefore, if an ongoing I/O operation is detected at the beginning of each cycle, the compensator can simply add $(T - \delta_i)/tdf_i$ to the container's virtual clock. However, since the I/O may finish during the pause, this design leads to a statistical inaccuracy, bounded by $(T - \delta_i)/tdf_i$. This bound reflects the granularity of the virtual time system. It is proportional to $\epsilon \times n/m$, where ϵ is the timescale, n is the number of containers, and m is the number of CPUs.

Our analysis shows that an I/O operation may cross multiple execution cycles, and the inaccuracy only occurs in the last cycle. Based on this observation, we improve the precision of I/O time compensation with the following design. For the cycles before the last one, we compensate for the I/O time in the aforementioned approach. In the last cycle, there are two scenarios. If the I/O finishes within the container's execution burst, no update is needed since the virtual clock has already resumed. Otherwise, we track the I/O completion time in the wall-clock time by modifying the system calls, such as `dio_bio_end_io` (a block I/O completion handler). The compensator now computes the exact I/O time elapsed in the last cycle and adds the time to the container's clock at the beginning of the next cycle. The detailed implementation is illustrated in Algorithm 2, and the evaluation results on VT-IO's ability to precisely control the virtual time during I/O are presented in Section 7.

Algorithm 1: Virtual Time Interface

```
Function updateVirtualTime (struct task_struct p)
    if p → isVirtual == 1 then
        wallClockTime = now();
        runningTime = wallClockTime − p → timeToUpdate;
        dilatedRunningTime = runningTime/p → TDF;
        p → virtualTime += dilatedRunningTime;
        p → timeToUpdate = wallClockTime;
    end
end
Function new_gettimeofday (struct timeval tv)
    if current → isVirtual == 1 then
        updateVirtualTime(current);
        tv = ns_to_timeval(current → virtualTime);
    else
        gettimeofday(tv);
    end
end
```

6.2 Precise Virtual Time Synchronization

Applying the I/O time compensator may lead to the inconsistency of virtual time between the containers with and without the compensation mechanism. The processes that trigger an I/O time compensation may have a 'false start' in the next execution cycle because their virtual clocks are ahead of the clocks of other CPU-intensive processes. The inconsistency may lead to causality issues among the events generated by the processes in different containers. To fix this issue, we implement an execution time adapter that dynamically adjusts the length of the execution time of each container with a false start. At the beginning of each execution cycle, the synchronization controller checks the virtual clock for each container. If one is ahead of the others, the synchronization controller adjusts the scheduling of the next cycle by reducing the execution burst of the false-started container. Therefore, the virtual clock of other processes can gradually catch up over cycles. As shown in `executionTimeAdapter()` in Algorithm 2, the adapter dynamically updates the execution time in each cycle. Once the adapter finds a 'false start' process, the process's execution time is reduced during the following cycles until `falseStart` reaches zero.

7 SYSTEM EVALUATION

We evaluate the performance of VT-IO in terms of accuracy during I/O operations, synchronization overhead, and system scalability. The experiments are conducted on a 64-bit Linux platform (Ubuntu 14.04 with a modified Linux Kernel). The machine has two 64-Core processors, 1 TB RAM, a 12-TB hard disk drive with a sustained data transfer rate of 248 MB/s, and Nvidia Quadro P400 GPU. Each experiment is repeated at least 10 times.

7.1 I/O Temporal Accuracy

The ability to precisely advance the virtual clocks during I/O operations and GPU computation distinguishes VT-IO from the other container-based virtual time systems. We perform three experiments with GPU tasks, network I/O, and disk I/O and compare the temporal accuracy of VT-IO with TimeKeeper. We measure the virtual time elapsed for each scenario: (a) time to perform 10,000 iterations of matrix additions on GPU, (b) round-trip

Algorithm 2: Time Compensator

```
Function compensateIOTime (struct task_struct p)
    if p → compensationFlag == 1 then
        // Compensation I/O time and reset flag
        wallClockTime = now();
        if p → ioCompleteTime > 0 then
            IORunningTime = p → ioCompleteTime − p → timeToUpdate;
        else
            IORunningTime = wallClockTime − p → timeToUpdate;
        end
        compensationTime = IORunningTime/p → TDF;
        p → compensationFlag = 0;
        p → ioCompleteTime = 0;
        p → timeToUpdate = wallClockTime;
    end
end
Function executionTimeAdapter (struct task_struct p)
    // Execution time as it is defined in Equation 1
    execTime = QUANTA × (p → tdf)/(leader → tdf);
    if p → falseStart > 0 then
        // Reduce the execution time to mitigate the effect of false start
        if p → falseStart > execTime then
            execTime = 0;
            p → falseStart −= execTime;
        else
            execTime −= p → falseStart;
            p → falseStart = 0;
        end
    end
    return execTime;
end
```


time of a UDP communication with a one-second link delay, and (c) time to direct write 100MB data to disk. In order to limit the influence of resource completion, only one container conducts the I/O operations while the other processes are running CPU-intensive applications like sysbench [4]. Thus, the wall clock time elapsed during the I/O remains the same regardless of the number of containers. We plot the physical testbed measurements, which serve as the ground truth, as well as the results of TimeKeeper and VT-IO in Figure 13 for all three sets of experiments.

We observe in all three scenarios (GPU, network I/O, and disk I/O), that the virtual time in TimeKeeper decays when the number of containers increases by m, where m is the number of cores. The cause of such an error is modeled and analyzed in Section 3. On the other hand, VT-IO successfully maintains the correct virtual time as the number of containers increases. The results are very close to the physical testbed measurements, as shown

Algorithm 3: Execution Control

```
Function resume (struct task_struct p)
    if p → compensationFlag == 1 then
        compensateIOTime(p);
    else
        p → timeToUpdate = now();
    end
    // Start execution on CPU
    kill(p, SIGCONT);
end
Function pause (struct task_struct p)
    updateVirtualTime(p);
    // Set flag if I/O not finished
    if isBlockedForIO(p) then
        compensationFlag = 1;
    end
    kill(p, SIGSTOP);
end
```

in Figure 13. Without the time compensation mechanism, the error of I/O can be significant. For instance, in the experiment with 30 containers, the error caused by a network I/O operation is up to 87.31% in TimeKeeper compared with the physical testbed, while the error is less than 3.6% in VT-IO. Note that the standard deviations of measurement in Figure 13a and Figure 13b ranged from 0.0012 seconds to 0.0028 seconds, which are hard to observe on the plots.

7.2 Scalability

To study the scalability of VT-IO with DLM, we first fix the number of CPUs to 32 and 126 and measure the execution time per cycle with various numbers of containers. The results are plotted in Figure 14. We observe that the execution time linearly increases as the number of containers grows. We then fix the number of containers to 64, 256, and 512 and measure the execution time per cycle with various numbers of CPUs. The results are plotted in Figure 15. We observe that the parallel execution of containers is efficient since the execution time dramatically decreases as the number of CPUs increases. The standard deviations in Figure 14 and Figure 15 are two orders of magnitude less than the mean value and thus are hard to observe on the plots.

The results indicate that both models exhibit similar runtime for each execution cycle. For example, when using a configuration of 32 cores and 512 containers, the runtime per cycle for VT-IO is 0.1607 seconds, while the model with DLM is 0.1614 seconds, resulting in a difference of only 0.4%. Detailed results can be found in Figures 14 and 15, demonstrating good and comparable scalability in both models. This can be attributed to the independent operation of each container, eliminating the need for data synchronization mechanisms such as locks in multi-thread parallelism, which would introduce additional overhead.

7.3 Synchronization Overhead

Similar to Timekeeper and other virtual time systems, VT-IO with DLM introduces the synchronization overhead due to the following two reasons: 1) the time spent to wake up kernel threads to start synchronization; note

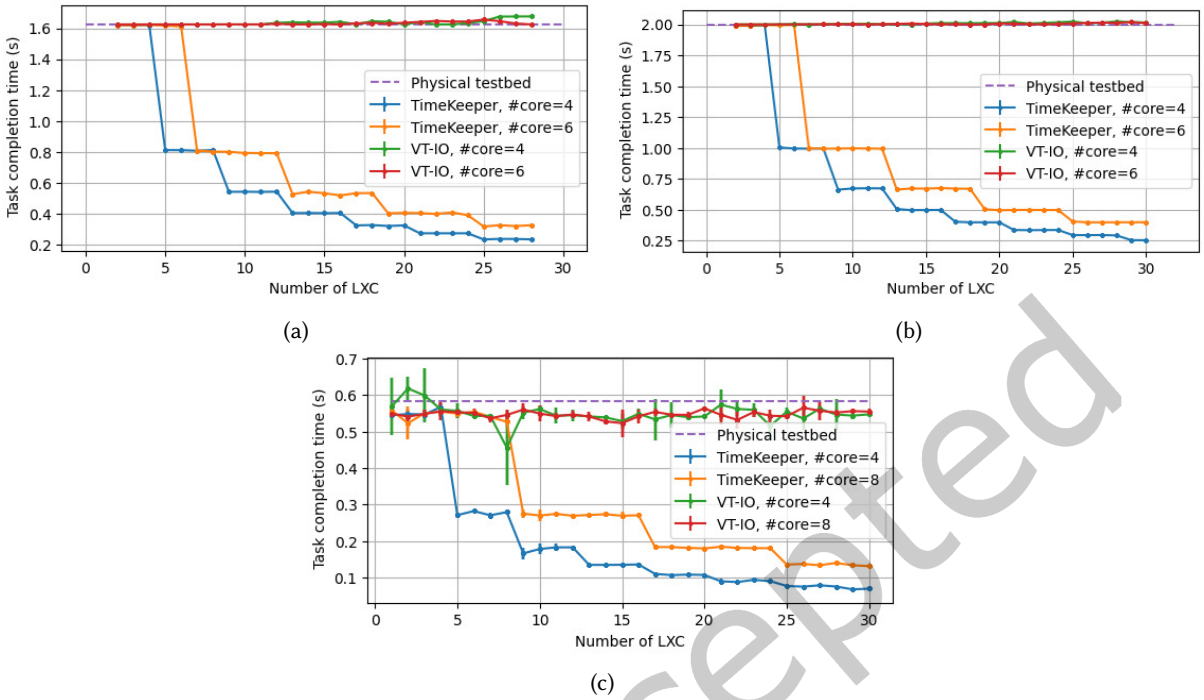


Fig. 13. Measurement of virtual time elapsed during (a) matrix additions on GPU, (b) round-trip time of socket communication, and (c) direct disk write on a physical testbed (i.e., ground-truth measurement), Timekeeper and VT-IO

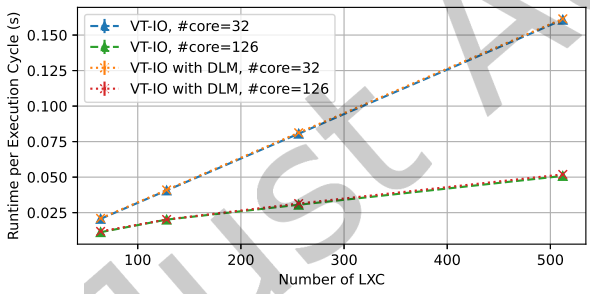


Fig. 14. Execution time per cycle vs. number of LXCs

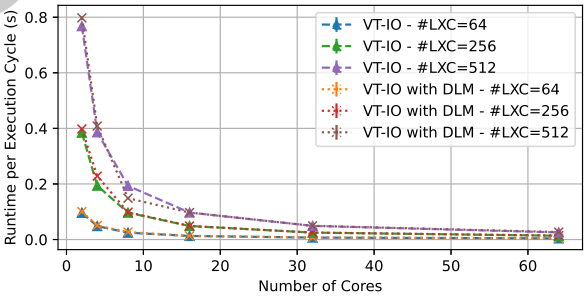
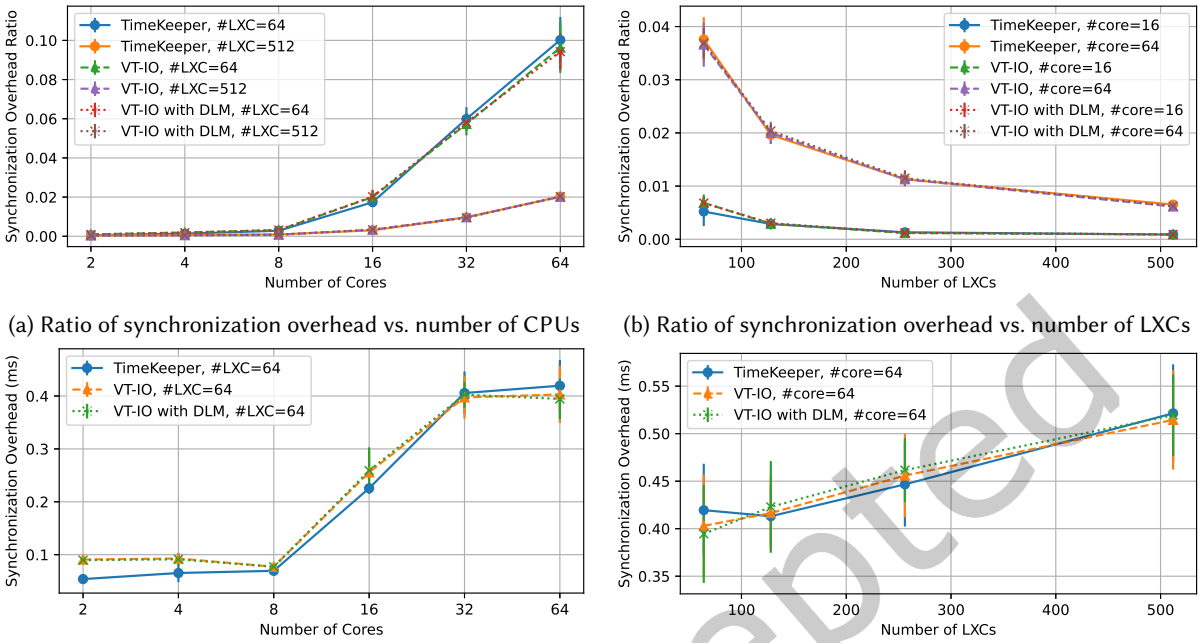


Fig. 15. Execution time per cycle vs. number of CPUs

that each CPU is associated with a kernel thread, and 2) the computational time for each kernel thread to complete task scheduling (e.g., execution Time Adapter in Algorithm 2). We conduct experiments to measure the synchronization overhead for each execution cycle and compare the overhead among TimeKeeper, VT-IO, and VT-IO with DLM.

We compute the ratio of synchronization overhead in TimeKeeper, VT-IO, and VT-IO with DLM by varying the number of CPUs and containers. The ratio of synchronization overhead is defined as the synchronization time over the execution time in one cycle. The results are plotted in Figure 16a and Figure 16b respectively. We



(a) Ratio of synchronization overhead vs. number of CPUs

(b) Ratio of synchronization overhead vs. number of LXCs

(c) Synchronization overhead time vs. number of CPUs

(d) Synchronization overhead time vs. number of LXCs

Fig. 16. Synchronization overhead on VT-IO, VT-IO with DLM, and TimeKeeper

observe that the overhead introduced by our I/O time compensator and dynamic load monitor are very minimal compared with TimeKeepr and VT-IO as shown in Figure 16a and Figure 16b. For instance, the overhead ratio of 512 containers on 64 CPUs is 2.026% on TimeKeeper, 2.011% on VT-IO, and 2.031% on VT-IO with DLM. Figure 16c shows the relation between the time of synchronization overhead and the number of CPUs. The synchronization overhead doesn't increase as the number of CPUs increases from 2 to 8 because of the reduced workload on each thread. As the number of CPUs keeps increasing from 8 to 64, the cost of controlling the kernel threads now dominates the overhead. As shown in Figure 16c, the synchronization overhead of TimeKeeper and VT-IO are close. For instance, the overhead to emulate 64 containers on 32 CPUs in VT-IO with DLM is 0.4025 ms in each cycle, while the overhead is 0.4056 ms in TimeKeeper and 0.3973 ms in VT-IO. The difference is within 0.01 ms. Figure 16d shows the relation between the time of synchronization overhead and the number of containers. The overhead increases linearly as the number of containers grows, and more containers lead to an increased workload of task scheduling. However, Figure 16b shows that the ratio of synchronization overhead drops even if the number of containers increases. It is because the execution time increases linearly as the number of containers grows (see Figure 14), and the improvement in execution time is more significant than the cost of the increased overhead, and thus results in performance gain with the increasing number of containers. VT-IO with DLM experiences a similar synchronization overhead compared with TimeKeeper and VT-IO. As shown in Figure 16d, to emulate 512 containers on 64 CPUs, the overhead introduced by VT-IO with DLM is 0.5194 ms, while the overhead is 0.5144 ms in VT-IO and 0.5214 in TimeKeeper. The difference is less than 0.9%.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we discover and analyze the virtual time advancement error in existing virtual time systems because of I/O operations. We develop VT-IO with Dynamic Load Monitor that provides precise time advancement control during disk I/O, network I/O, and GPU operations even during the I/O resource contention among multiple processes. We conduct a comparative evaluation with TimeKeeper and a physical testbed to show that VT-IO is capable of maintaining high temporal fidelity during I/O operations with limited synchronization overhead. Our future work aims to further speed up the virtual-time-enabled emulation testbed. One approach is to integrate the scheduler with a load balancer, which is responsible for optimizing the process allocation with limited overhead. We will also integrate VT-IO to a network simulator to improve the flexibility and scalability of the testbed.

ACKNOWLEDGMENTS

The authors are grateful for the support of the National Science Foundation (NSF) under Grant CNS-2247721, CNS-2034870, and the NSF Center for Infrastructure Trustworthiness in Energy Systems (CITES) under Grant EEC-2113903. The authors also express their gratitude for the financial support provided by the Chancellor's Fund for Innovation and Collaboration at the University of Arkansas.

REFERENCES

[1] 2005. OpenVZ: a container-based virtualization for Linux. https://openvz.org/
[2] 2021. OpenDSS: an electric power distribution system simulator. https://www.epri.com/pages/sa/opendss
[3] 2022. Mininet: An Instant Virtual Network on your Laptop (or other PC). http://mininet.org/
[4] 2022. SysBench: a cross-platform and multi-threaded benchmark tool for evaluating OS parameters. http://manpages.ubuntu.com/manpages/trusty/man1/sysbench.1.html
[5] 2023. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page
[6] 2023. Ptrace: The linux process tracing subsystem. https://man7.org/linux/man-pages/man2/ptrace.2.html
[7] 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.*
[8] Vignesh Babu and David Nicol. 2022. Temporally synchronized emulation of devices with simulation of networks. In *Proceedings of the 2022 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation.*
[9] 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).
[10] Daniel Pierre Bovet, Marco Cassetti, and Andy Oram. 2000. *Understanding the Linux Kernel.* O'Reilly & Associates, Inc.
[11] Gong Chen, Zheng Hu, and Dong Jin. 2022. Integrating I/O Time to Virtual Time System for High Fidelity Container-based Network Emulation. In *Proceedings of the 2022 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation.*
[12] 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.*
[13] Thomas Gleixner and Douglas Niehaus. 2006. Hrtimers and Beyond: Transforming the Linux Time Subsystems. In *Proceedings of the Ottawa Linux Symposium.*
[14] 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).
[15] 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.*
[16] Christopher Hannon, Jiaqi Yan, and Dong Jin. 2016. DSSnet: A Smart Grid Modeling Platform Combining Electrical Power Distribution System Simulation and Software Defined Networking Emulation. In *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation.*
[17] Christopher Hannon, Jiaqi Yan, Yuan-An Liu, and Dong Jin. 2019. A distributed virtual time system on embedded Linux for evaluating cyber-physical systems. In *Proceedings of the 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation.*
[18] Matt Helsley. 2009. LXC: Linux container tools. https://developer.ibm.com/tutorials/l-lxc-containers/
[19] Dong Jin, Yuhao Zheng, Huaiyu Zhu, David M Nicol, and Lenhard Winterrowd. 2012. Virtual time integration of emulation and parallel simulation. In *Proceedings of 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation.* 201–210.
[20] 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.*

[21] Jason Liu. 2008. A Primer for Real-Time Simulation of Large-Scale Networks. *41st Annual Simulation Symposium* (2008), 85–94.
[22] Robert Love. 2010. *Linux Kernel Development* (3rd ed.). Addison-Wesley Professional.
[23] Pratik Mishra and Arun K. Somani. 2017. Host managed contention avoidance storage solutions for Big Data. *J. Big Data* 4, 1 (June 2017), 18.
[24] David M. Nicol. 1993. The Cost of Conservative Synchronization in Parallel Discrete Event Simulations. *J. ACM* (1993).
[25] David M. Nicol, Dong Jin, and Yuhao Zheng. 2011. S3F: The Scalable Simulation Framework Revisited. In *Proceedings of the Winter Simulation Conference*.
[26] Marie-Therese Puth, Markus Neuhäuser, and Graeme D. Ruxton. 2014. Effective use of Pearson's product–moment correlation coefficient. *Animal Behaviour* 93 (2014), 183–189.
[27] George F. Riley and Thomas R. Henderson. 2010. *The ns-3 Network Simulator*. Springer Berlin Heidelberg.
[28] Patrick Schober, Christa Boer, and Lothar A. Schwarte. 2018. Correlation Coefficients: Appropriate Use and Interpretation. *Anesthesia & Analgesia* 126, 5 (2018).
[29] Brian Walters. 1999. VMware Virtual Platform. *Linux J.* (1999).
[30] Jon Watson. 2008. VirtualBox: Bits and Bytes Masquerading as Machines. *Linux J.* (2008).
[31] 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*.
[32] Gavin Wood. 2022. Ethereum: A secure decentralised generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf
[33] Xiaoliang Wu, Jiaqi Yan, and Dong Jin. 2019. Virtual-Time-Accelerated Emulation for Blockchain Network and Application Evaluation. In *Proceedings of the 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*.
[34] Jiaqi Yan and Dong Jin. 2015. VT-Mininet: Virtual-Time-Enabled Mininet for Scalable and Accurate Software-Define Network Emulation. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*.
[35] Xuanxia Yao, Peng Geng, and Xiaojiang Du. 2013. A Task Scheduling Algorithm for Multi-core Processors. In *Proceedings of the 2013 International Conference on Parallel and Distributed Computing, Applications and Technologies*.
[36] Yuhao Zheng and David M. Nicol. 2011. A Virtual Time System for OpenVZ-Based Network Emulations. In *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*.