Integrating I/O Time to Virtual Time System for High Fidelity Container-based Network Emulation

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

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

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

ABSTRACT

Container-based network emulation provides an accurate, flexible, and cost-effective application design and evaluation testing environment. Enabling virtual time to processes running inside containers essentially improves the temporal fidelity of emulation experiments. However, the lack of precise time management during operations, such as disk I/O, network I/O, and GPU computation, often leads to virtual time advancement errors observed in existing virtual time systems. This paper proposes VT-IO, 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. VT-IO enables accurate virtual time advancement with precise I/O time measurement and compensation. The experimental results show that the temporal error can be reduced from 87.31% to 3.6% and VT-IO only introduces around 2% overhead of the total execution time. Finally, we demonstrate VT-IO's usability and temporal fidelity improvement with a case study of a Bitcoin mining application.

CCS CONCEPTS

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

KEYWORDS

Virtual Time, Network Emulation, Linux Container, TimeKeeper, Virtualization

ACM Reference Format:
Gong Chen, Zheng Hu, and Dong Jin. 2022. Integrating I/O Time to Virtual Time System for High Fidelity Container-based Network Emulation. In *SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '22), June 8–10, 2022, Atlanta, GA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3518997.3531023

SIGSIM-PADS '22, June 8–10, 2022, Atlanta, GA, USA

ACM ISBN 978-1-4503-9261-7/22/06…$15.00
https://doi.org/10.1145/3518997.3531023

1 INTRODUCTION

The networking industry is evolving day by day. Rapid innovation highly relies on the successful transformation from early-stage research ideas to real-world applications. A high-fidelity and flexible testing environment is often an essential tool to realize such transformation. Network simulation testbeds, on the one hand, provide good flexibility and scalability but raise a concern about fidelity due to model simplification and abstraction. Physical testbeds, on the other hand, allow real-time and high-fidelity experiments on real devices, but the scale is limited as it is often too expensive or even infeasible to build the physical testbed. To balance the scalability and fidelity, researchers also explore the container-based network emulation testbed, which constructs tens or hundreds of containers on a single physical machine. Each container represents a virtual node (e.g., end-host, router, switch, or middlebox) and allows unmodified code execution in the container to preserve fidelity.

Network emulators provide rapid prototyping of network applications by leveraging virtualization techniques, such as Xen [7], OpenVZ [1], and Linux Containers (LXC) [16]. While executing binaries instead of abstract models ensures functional fidelity, a network emulator may still face temporal inaccuracy. It 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 [5, 7, 12, 13, 15, 17, 25, 27, 29]. 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 [17]. 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 extensive 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 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.

We first empirically demonstrate the limitations of the existing virtual time systems (e.g., TimeKeeper) during I/O operations. We then analyze the root cause by mathematically modeling the temporal error, and the analytical results match well with the experimental results. 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. VT-IO controls containers' execution and time synchronization on both CPU and I/O activities. With a minimal modification to the Linux kernel, VT-IO enables tracking of each process's I/O states. At the beginning of each cycle, VT-IO checks if an unattended I/O exists and compensates the time elapsed during this I/O operation to the virtual clock of the process. We evaluate the performance 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. The temporal behaviors match the physical testbed measurements for disk I/O, network I/O and GPU experiments conducted on VT-IO. VT-IO significantly 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 8 CPUs is less than 0.32 ms per cycle which is around 0.33% of the overall execution time. The execution time of the same experiment linearly decreases as the number of containers grows. Meanwhile, VT-IO maintains a stable and high precision of I/O time even with a large number of containers. For instance, VT-IO yields a mean absolute error of around 6 ms regardless of the number of CPUs and containers. Finally, we present a case study of a blockchain application that runs intensive GPU tasks and demonstrate the usability and temporal fidelity improvement of VT-IO.

The remainder of the paper is organized as follows. Section 2 models and analyzes the temporal error of a virtual time system during I/O operations. Section 3 describes the architecture of VT-IO, a virtual time system that integrates precise I/O time management. Section 4 illustrates the implementation details of VT-IO, including synchronization and I/O time compensation. Section 5 evaluates VT-IO in terms of accuracy, synchronization overhead, and scalability. Section 6 demonstrates the usability of VT-IO through a blockchain emulation application. Section 7 presents the related works. Section 8 concludes the paper with future work.

2 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 [7, 12, 13, 15, 17, 25, 27, 29] that rely on the operating system's clock to control the execution of emulated processes and 2) instruction-count based approaches [5, 6] 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 non-CPU-intensive jobs to complete, such as disk I/O, network I/O, or GPU computation. We use TimeKeeper [17] as a demonstrative case since many recent works [5, 15, 26, 27], 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.

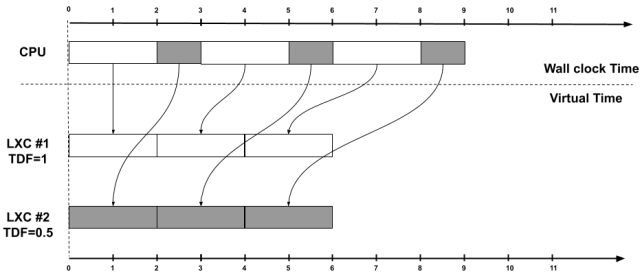


Figure 1: Virtual time advancement for CPU intensive processes

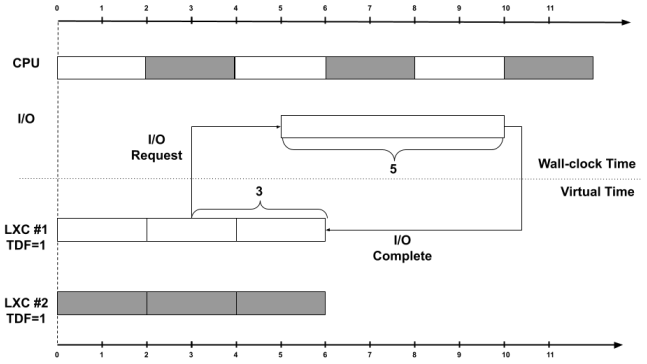


Figure 2: Virtual time advancement with I/O operations in Timekeeper. In this example, I/O time is 5 seconds and thus *LXC* #1 should advance its virtual time for 5 seconds instead of 3 seconds (tdf = 1), i.e., no integration of I/O time in TimeKeeper causes a temporal error of 2 seconds.

2.1 Virtual Time Advancement in TimeKeeper

TimeKeeper is a lightweight virtual time system for Linux container (LXC) [16] based network emulation. While full virtualization (e.g., VMWare [23], VirtualBox [24]) and paravirtualization (e.g., Xen [7]) 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) [14], 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.

TimeKeeper proposed a barrier-based conservative synchronization mechanism [20] 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 to 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 offer 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.

2.2 Demonstration of TimeKeeper Imprecision with I/O Intensive Tasks

TimeKeeper and other timer-based virtual time systems [5, 15, 26, 27] 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 it is paused. As shown in Figure 2, from time 6 to 8, it is *LXC*#2's turn to be executed on CPU while *LXC*#1 is paused. As a result, *LXC*#1's clock does not advance during 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.

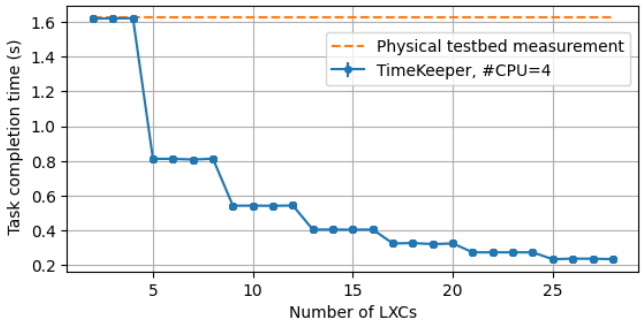


Figure 3: Demonstration of temporal error: Measurement of the GPU task completion time in TimeKeeper

We empirically demonstrate the timing imprecision and show the experimental result in Figure 3. Only one container conducts non-CPU tasks in the experiment, while the others run CPU-intensive applications. For demonstration purposes, the non-CPU 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 5. 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 red 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 grow. We discovered an interesting pattern that the completion time drops when the number of containers increases by four. In fact, four is also the number of CPUs used in the experiment. The following section mathematically models the error to explain the observed pattern.

2.3 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.

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

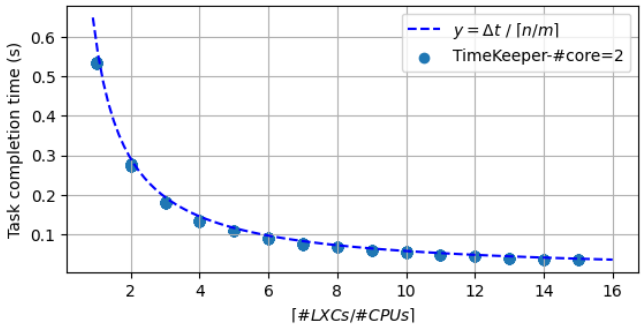


Figure 4: Measurement of disk write operation completion time in TimeKeeper. n is the number of containers. m is the number of CPUs. Δt is measurement of time elapsed during write operation in wall clock. TDF of all containers in this scenario is 1.

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.

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

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.

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

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.

2.4 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 complete 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.

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

To mitigate the temporal error during I/O, we design 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 I/O time is discussed in detail in Section 3.3 and Section 4.1.

Synchronization Controller is designed to synchronize virtual time among the containers (see Section 3.2 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 4.2.

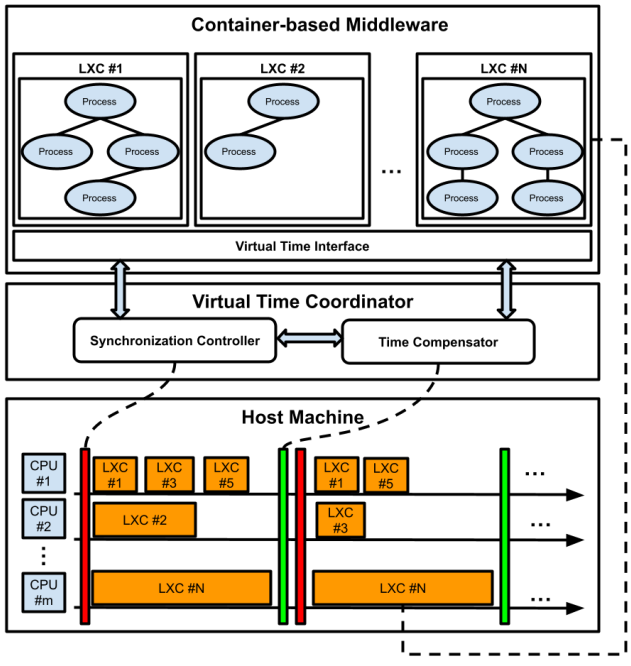


Figure 5: Architecture design of VT-IO

3 ARCHITECTURE

We design a virtual time system named VT-IO that integrates precise 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) Virtual Time Coordinator that controls the execution and time synchronization of containers on CPU and I/O activities. The architecture design of VT-IO is illustrated in Figure 5.

3.1 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 containers 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 4.

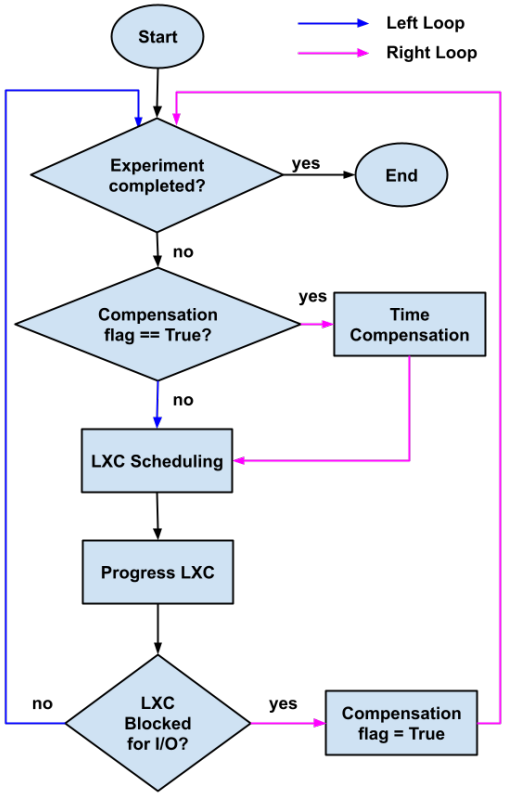


chart.png

Figure 6: Workflow of I/O time compensator in an execution cycle

3.2 Virtual Time Coordinator with I/O Time Compensator

It's 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., available CPUs, network bandwidth) on the physical machine are insufficient to support a large number of containers [27].

To address this problem, researchers have explored the virtual machine scheduling mechanism [25, 29], based on which existing works have proposed various designs for Xen [12–14] and Linux Container [5, 17, 29]. 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 two barrier-based modules: Synchronization Controller and Time Compensator, 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 [28], 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.

Figure 6 shows the workflow of the Time Compensator to handle the above issue. Two loops are presented in Figure 6. The left loop describes the workflow when an LXC is not blocked for I/O at the

end of a cycle. The right loop describes how Time Compensator updates the virtual clock of a container with an uncompleted I/O as well as collaborates with the Synchronization Controller to adjust the scheduling for the next cycle since the virtual clock of the updated container may be ahead of the other containers due to the compensation.

Figure 7 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 update *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.

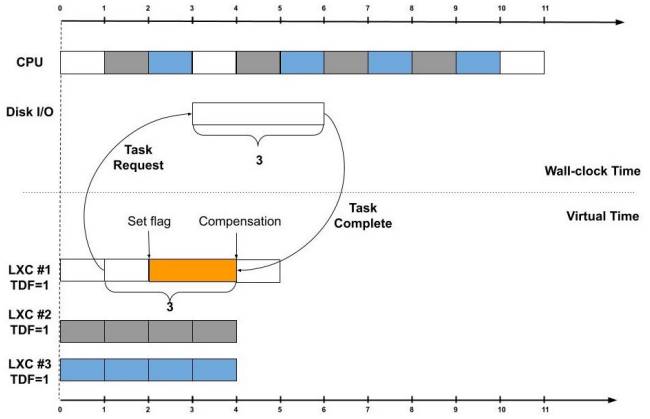


Figure 7: Virtual time advancement with I/O operations in VT-IO

4 IMPLEMENTATION

We modify the Linux kernel to implement a virtual time system called VT-IO based on the architecture design shown in Figure 5. Container-based Middleware intercepts the timing-related system calls in Linux to enable virtual time of the containers. Similar to the existing container-based virtual systems [5, 17], we modify the data structure `task_struct`, which is a process descriptor containing every relevant information about a process [18]. 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.

We use signaling technology, such as SIGSTOP and SIGCONT, and hrtimers [11], and a high-resolution timer provided by the mainstream Linux kernel, to control the process execution in the containers. Algorithm 2 illustrates the time compensation procedure. The `pause()` and `resume()` functions describe how to pause and resume a process on CPU and how to update its virtual clock. Different from TimeKeeper [17], 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.

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
```

4.1 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 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 5.

4.2 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.

5 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

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
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
```

data transfer rate of 248 MB/s, and Nvidia Quadro P400 GPUs. Each experiment is repeated at least 10 times.

5.1 I/O Temporal Accuracy

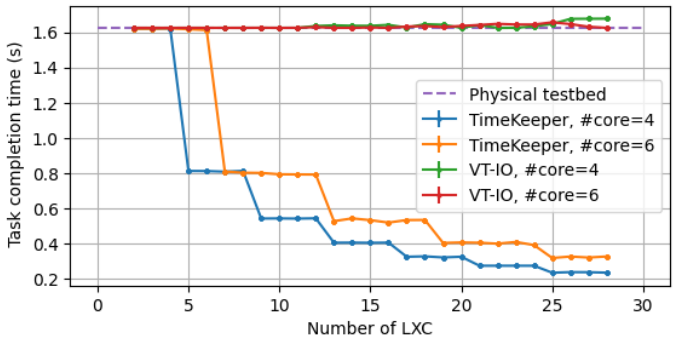
The ability to precisely advance the virtual clocks during I/O operations and GPU computation distinguishes VT-IO from the other container-based virtual time system. We perform three experiments with GPU tasks, network I/O, and disk I/O and compare the temporal accuracy of VT-IO with TimeKeeper. We measure the virtual time elapsed for each scenario: (a) time to perform 10,000 iterations of matrix additions on GPU, (b) round-trip time of a UDP communication with 1-second link delay, and (c) time to 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 the 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 8 for all three sets of experiments.

We observe in all three scenarios (GPU, network I/O, and disk I/O), the virtual time in TimeKeper decays when the number of containers increases by m, where m is the number of cores. The cause of such error is modeled and analyzed in Section 2.2 and 2.3. 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 in Figure 8. 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 8a and Figure 8b are ranged from 0.0012 seconds to 0.0028 seconds, which are hard to observe on the plots.

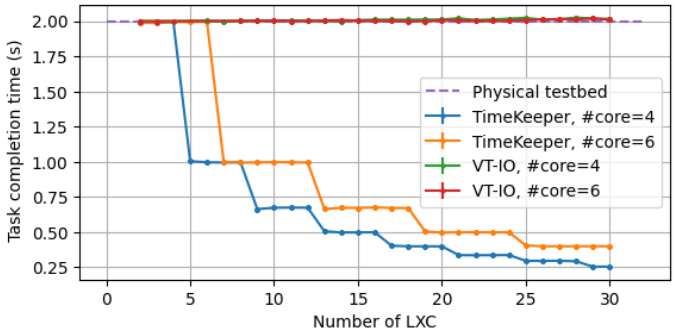
5.2 Scalability

To study the scalability of VT-IO, we first fix the number of CPUs to 2, 8, and 16 and measure the execution time per cycle with various numbers of containers. The results are plotted in Figure 9. We observe that the execution time linearly increases as the number of containers grows. The overhead introduced by VT-IO is minimal, e.g., approximately 2% for 512 containers on 64 CPUs. 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 10.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 9 and Figure 10 are two orders of magnitude less than the mean value and thus are hard to be observed on the plots.

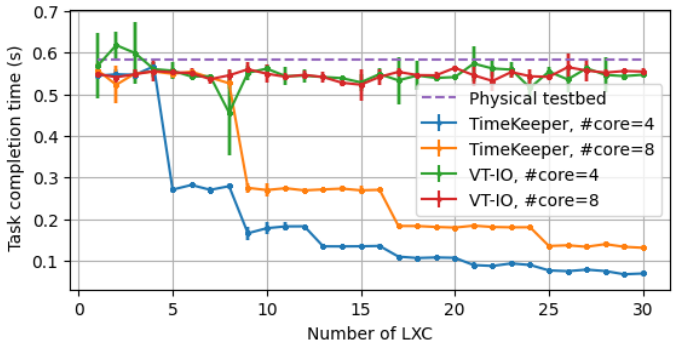
VT-IO maintains a high and stable time precision even with a large number of containers. We write 10M data to a disk and measure the I/O time in TimeKeeper, VT-IO, as well as a physical machine as the ground truth. We evaluate the virtual time systems using Mean Absolute Error (MAE), defined as the time difference between the virtual time system and the physical testbed. As shown in Figure 11, VT-IO maintains a much lower MAE of 6 ms while



(a)



(b)



(c)

Figure 8: Measurement of virtual time elapsed during (a) matrix additions on GPU, (b) round-trip time of socket communication, and (c) disk write on a physical testbed (i.e., ground-truth measurement), Timekeeper and VT-IO

TimeKeeper introduces an error of up to 61.5 ms with 512 containers on 4 CPUs. The error introduced by VT-IO is bounded by a mean error of 5.19 ms and is relatively stable regardless of the number of containers and CPUs. However, the error introduced by TimeKeeper increases significantly with the growing number of containers, which varies from 1.23 ms up to 61.90 ms. In addition, emulation using fewer CPUs yields more significant errors in TimeKeeper. For instance, the emulation of 64 containers with 16 CPUs yields an average error of around 41.99 ms in disk write I/O, while the emulation with 4 CPUs has an error of around 59.12 ms, which is

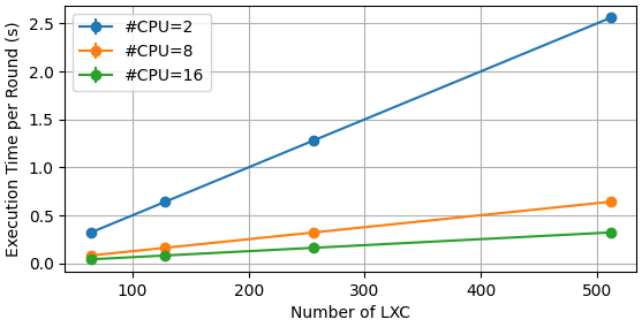


Figure 9: Execution time per cycle vs. number of LXCs

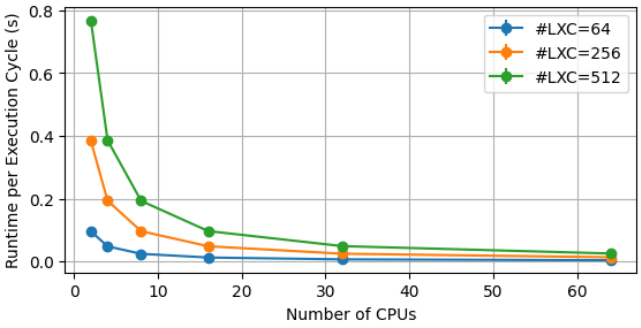


Figure 10: Execution time per cycle vs. number of CPUs

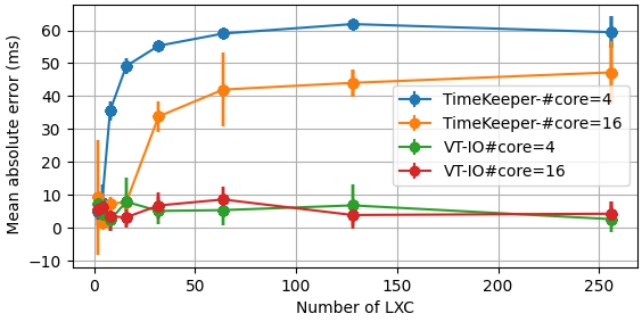


Figure 11: Mean absolute error (MAE) for disk write operation time

40.79% more. The root cause has been explained in Section 2.2 and Section 2.3.

5.3 Synchronization Overhead

Similar to Timekeeper and other virtual time systems, VT-IO introduces the synchronization overhead due to the following two reasons: 1) the time spent to wake up kernel threads to start synchronization; note that each CPU is associated with a kernel thread, and 2) the computational time for each kernel thread to complete task scheduling (e.g., exectuionTimeAdapter in Algorithm 2). We conduct experiments to measure the synchronization overhead for each execution cycle and compare the overhead with TimeKeeper.

We compute the ratio of synchronization overhead in TimeKeeper and VT-IO 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 12 and Figure 13 respectively. We observe that the overhead introduced by our I/O time compensator is very minimal compared with TimeKeepr as shown in Figure 12 and Figure 13. For instance, the overhead ratio of 512 containers on 64 CPUs is 2.026% on TimeKeeper and 2.011% on VT-IO.

Figure 14 shows the relation between the time of synchronization overhead and the number of CPUs. The synchronization overhead decreases 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 14, the synchronization overhead of TimeKeeper and VT-IO are close. For instance, the overhead to emulate 64 containers on 32 CPUs in TimeKeeper is 0.474 ms in each cycle, while the overhead is 0.477 ms in VT-IO. The difference is only 0.003 ms.

Figure 15 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 13 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 9), 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 experiences a similar synchronization overhead compared with TimeKeeper. As shown in Figure 15, to emulate 256 containers on 8 CPUs, the overhead introduced by TimeKeeper is 0.1163 ms, while the overhead is 0.1167 ms in VT-IO. The difference is about four orders of magnitude less than the measurement.

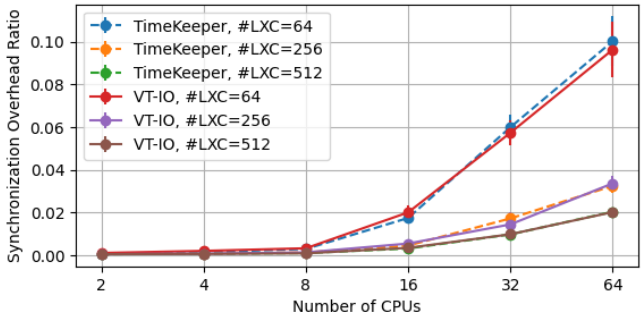


Figure 12: Ratio of synchronization overhead vs. number of CPUs

6 CASE STUDY

Nowadays, GPU is no longer just a configurable graphics processor but a programmable parallel processor. Applications are increasingly using GPU to solve intensive computing tasks in parallel, such as high-performance computing, deep learning, and cryptocurrency mining. Container-based emulation provides a flexible

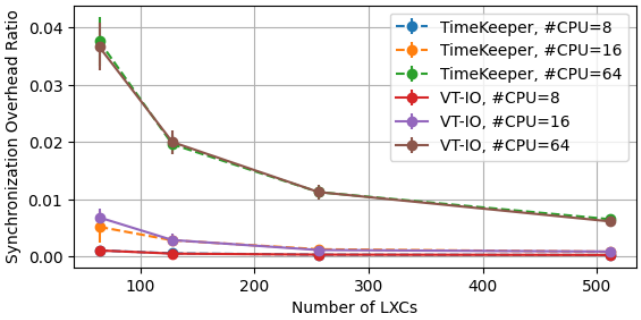


Figure 13: Ratio of synchronization overhead vs. number of LXCs

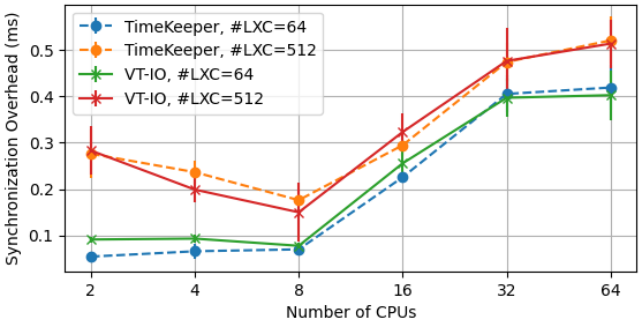


Figure 14: Synchronization overhead time vs. number of CPUs

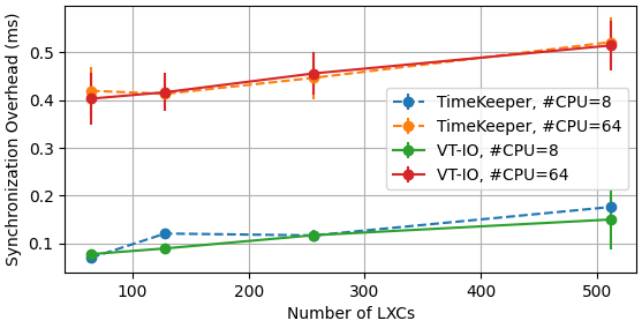


Figure 15: Synchronization overhead time vs. number of LXCs

and cost-effective way to test the design and implementation of those applications on a large scale. In this section, we demonstrate the usability of VT-IO through a case study of a Bitcoin mining application that demands massive GPU computational resources. We also perform a comparative evaluation on VT-IO, TimeKeeper, and a physical testbed to show the improvement of temporal fidelity in the virtual time system despite the intensive GPU computation.

6.1 Bitcoin Mining Experiment Setup

Bitcoin miners use powerful computing resources (e.g., GPU) to mine blocks, verify transactions, and record them on the blockchain [19]. The mining process is computing complex mathematical hash functions. Bitcoin network adjusts the difficulty of mining by raising or lowering the target hash to preserve a constant interval between new blocks. The difficulty, D_i in Equation 7, is defined as a measure of how difficult it is to mine Bitcoin blocks, i.e., finding a nonce of the first valid ith block whose hashed value is less than a target value. The difficulty is calculated using the following equation, where $Target_0$ is the target used in the Genesis Block and represents the initialized difficulty, and $Target_i$ is the current target.

$$D_i = \frac{Target_0}{Target_i} \tag{7}$$

Bitcoin's target value adjustment algorithm is expressed in Equation 8, where T_{expect} is a configurable constant that defines the expected time to complete mining of each block and T_i is the actual completion time.

$$Target_{i+1} = Target_i \times \frac{T_i}{T_{expect}} \tag{8}$$

In the traditional Bitcoin system, T_{expect} is set to be 10 minutes. As shown in Equation 9. Bitcoin system dynamically adjusts the difficulty based on the previous block's difficulty and computational time to regress to T_{expect}. Therefore, the speed of block generation is approximately 10 minutes regardless of the number of miners all over the world [22].

$$D_{i+1} = D_i \times \frac{T_{expect}}{T_i} \tag{9}$$

We conduct a set of emulation experiments of a Bitcoin mining application in VT-IO, TimeKeeper, and a physical testbed, respectively. The experiments are performed on a machine with 8 CPU cores and Nvidia Quadro P400 GPUs. There are 4 containers in each experiment, and the number of CPUs in use is 2. Only one of the containers runs the blockchain mining application on GPU, while the other containers run CPU-intensive applications (i.e., SysBench). Bitcoin difficulty changes every 2016 blocks in practice [9]. For demonstration purposes, we update the difficulty every 32 blocks to speed up the convergence in the experiment. T_{expect} is set to 0.5 seconds and the initial $Target_0$ is a random 256-bit integer, where the first 8 bits are zeros. The total number of blocks mined in the experiment is 2048.

6.2 Experimental Results

We conduct the experiments and observe that the difficulty converges for the physical testbed when the total number of mined blocks is 2048. We then plot the Bitcoin difficulty adjustment over time until 2048 mined blocks on all three testbeds in Figure 16. The x-axis is the index of each mined block in chronological order, and the y-axis is the difficulty value. The red dotted line presents the mean of the difficulty.

Bitcoin mining requires massive GPU calculations to add transactions to a blockchain. Due to the lack of virtual time management during GPU computation, the time spent on mining blocks in TimeKeeper appears to be less than the time in the physical testbed.

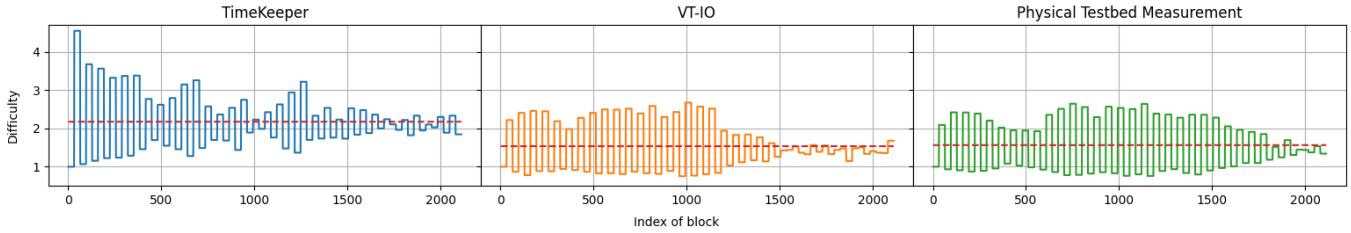


Figure 16: Bitcoin difficulty adjustment over time for TimeKeeper, VT-IO, and a physical testbed

Because of the precise virtual time control during GPU computation, with the same initial difficulty (i.e., target value) and block expected time, the difficulty adjustment in VT-IO matches well with the behavior in the physical testbed. VT-IO has a difficulty mean value of 1.54, which is close to the mean value of 1.56 in the physical testbed. However, Timekeeper has a much higher difficulty value of 2.18. The difficulty in TimeKeeper also fluctuates more than VT-IO and the physical testbed, i.e., the range is between 0.6 and 2.7 in VT-IO and the physical testbed, while the range is between 1.0 to 4.6 in TimeKeeper.

Please note that the hashes of blocks are randomly generated in our experiments, just like a real Bitcoin network. Therefore, it is reasonable not to have the same difficulty adjustment results across multiple runs as the mining time of each block is random. However, the computational power is stable in our experiments, and thus, the time to mine blocks converges eventually, while in a real Bitcoin network, more and more computers join to the mining, and hence the total computational power keeps growing [9].

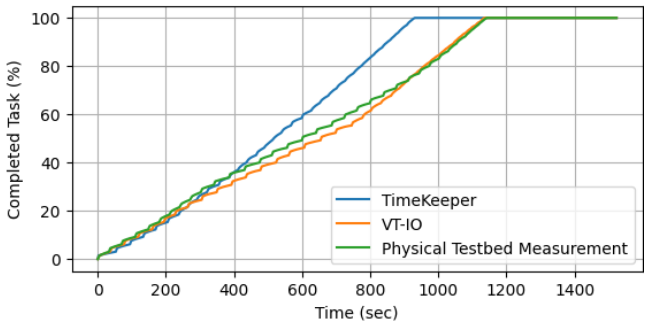


Figure 17: Accumulatively completed tasks over time

Figure 17 plots the percentage of the accumulatively completed tasks over time for all three testbeds. We observe that the performance of VT-IO is very close to the physical testbed. It takes 1142.46 seconds in the physical testbed and 1134.78 seconds in VT-IO to complete all the tasks. Meanwhile, TimeKeeper only spends 931.40 seconds. At 1000 seconds, the physical testbed accomplishes 83.01% of the tasks, and VT-IO accomplishes 84.48% of the tasks, while TimeKeeper has already accomplished 100% of the tasks.

7 RELATED WORK

7.1 Timer-based virtual time system

The timer-based virtual time system was proposed in [14]. The original idea was motivated by improving the scalability of OS virtualization in Xen [7]. Gupta et al. tried to slow down the clocks of VMs to mimic more powerful devices than what the underlying physical resources can offer (e.g., CPU, network bandwidth). The term Time Dilation Factor (TDF) indicates the relation between the wall clock time in the host machine and the virtual clock in the virtual machine, which is a key parameter of timer-based virtual time systems. Testbeds built on Xen, such as DieCast [13] and SVEET [10], are inspired by the design. For example, DieCast is a TDF-based approach to scale the performance of hardware, including CPU, network, and disk of virtual machines. It enables us to create a network emulation experiment on a physical machine with limited resources. However, the scheduling of VMs is managed by the VM hypervisor rather than the virtual time system in those approaches [10, 13, 14]. Although the VMs appears to run concurrently at a coarse time scale, the actual execution order and length are non-deterministic. Besides, the heavy-weight VMs limit the scale of a network that one can emulate. To address those limitations, Zheng et al. proposed a container-based network emulator [29]. By modifying the Linux kernel and leveraging a container-based virtualization technique, OpenVZ [1], they enabled precise and flexible control of the virtual clock's advancement. Jin et al. extended this work by developing a synchronization algorithm to integrate the container-based emulation, and a parallel network simulator [21]. Later, Lamps et al. proposed a Linux container-based virtual time system, TimeKeeper, which consists of fewer than 100 lines of code modification to the Linux Kernel [17]. It successfully emulated 45,000 containers on a machine with 32 CPUs, whose scale was two orders of magnitude larger than what OpenVZ can offer. More research work including VT-Mininet [27], Minichain [26], DSSNet [15] are inspired by TimeKeeper. However, due to the lack of precise time control over I/O operations, TimeKeeper suffers temporal inaccuracy when a process is waiting for tasks like disk I/O, network I/O and GPU computation to complete. To tackle this issue, we design and implement VT-IO to precisely compensate the I/O time in the virtual clock.

7.2 Instruction-based virtual time system

While most existing approaches derive the virtual time based on the system clock of host machines, Badu et al. proposed a new method to advance virtual time based on CPU instructions [5, 6]. Instead of controlling how long processes can be executed on the CPU

using hrtimer [11] and signaling techniques [8], Kronos controls processes' virtual time based on the binary instruction counts measured by ptrace [3], and perf [2]. 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 that 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 relative CPU speed to translate the instruction counts to virtual time. However, our analysis shows that Kronos also suffers from the same temporal error due to the lack of precise time control over I/O operations. Therefore, this motivates us to explore means to obtain the I/O operation time accurately and include that in the virtual time of each container.

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 and develop VT-IO that provides the precise time advancement control during operations, including disk I/O, network I/O, and GPU computation. We then 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. Finally, we demonstrate the usability and performance of VT-IO with a Bitcoin mining based network application and compare the results with TimeKeeper.

While VT-IO is capable of precisely measuring the time elapsed during I/O operations, the resources on the host machine, such as disk bandwidth, network bandwidth, and GPU, are distributed and shared among all the processes. If a large number of virtual hosts are simultaneously performing I/O operations, it may lead to an extensive resource competition that influences temporal fidelity. Our next step is to minimize the impact by adaptively adjusting the I/O time measurement based on the dynamic system load. Another direction is integrating the virtual time enabled emulator with simulators through efficient synchronization algorithms to create a high-fidelity and scalable testbed.

ACKNOWLEDGMENTS

This work is partly sponsored by the NSF Center for Infrastructure Trustworthiness in Energy Systems (CITES) under Grant EEC-2113903 and the Maryland Procurement Office under Contract No. H98230-18-D-0007. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF and the Maryland Procurement Office.

REFERENCES

[1] 2005. OpenVZ: a container-based virtualization for Linux. https://openvz.org/
[2] 2020. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page
[3] 2021. Ptrace: The linux process tracing subsystem. https://man7.org/linux/man-pages/man2/ptrace.2.html
[4] 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] 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.*
[6] Vignesh Babu and David M. Nicol. 2018. On Repeatable Emulation in Virtual Testbeds. In *Proceedings of the 2018 Winter Simulation Conference.*
[7] 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).
[8] Daniel Pierre Bovet, Marco Cassetti, and Andy Oram. 2000. *Understanding the Linux Kernel.* O'Reilly & Associates, Inc., USA.
[9] Rainer Böhme, Nicolas Christin, Benjamin Edelman, and Tyler Moore. 2015. Bitcoin: Economics, Technology, and Governance. *Journal of Economic Perspectives* 29, 2 (May 2015), 213–38.
[10] 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.*
[11] Thomas Gleixner and Douglas Niehaus. 2006. Hrtimers and Beyond: Transforming the Linux Time Subsystems. In *Proceedings of the Ottawa Linux Symposium.*
[12] Andreas Grau, Steffen Maier, Klaus Herrmann, and Kurt Rothermel. 2008. Time Jails: A Hybrid Approach to Scalable Network Emulation. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation.*
[13] 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).
[14] 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.*
[15] 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.*
[16] Matt Helsley. 2009. LXC: Linux container tools. https://developer.ibm.com/tutorials/l-lxc-containers/
[17] 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.*
[18] Robert Love. 2010. *Linux Kernel Development* (3rd ed.). Addison-Wesley Professional.
[19] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system http://bitcoin.org/bitcoin.pdf.
[20] David M. Nicol. 1993. The Cost of Conservative Synchronization in Parallel Discrete Event Simulations. *J. ACM* (1993).
[21] David M. Nicol, Dong Jin, and Yuhao Zheng. 2011. S3F: The Scalable Simulation Framework Revisited. In *Proceedings of the Winter Simulation Conference.*
[22] Shunya Noda, Kyohei Okumura, and Yoshinori Hashimoto. 2020. An Economic Analysis of Difficulty Adjustment Algorithms in Proof-of-Work Blockchain Systems. In *Proceedings of the 21st ACM Conference on Economics and Computation.*
[23] Brian Walters. 1999. VMware Virtual Platform. *Linux J.* (1999).
[24] Jon Watson. 2008. VirtualBox: Bits and Bytes Masquerading as Machines. *Linux J.* (2008).
[25] 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.*
[26] 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.*
[27] 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.*
[28] Xuanxia Yao, Peng Geng, and Xiaojiang Du. 2013. A Task Scheduling Algorithm for Multi-core Processors. In *2013 International Conference on Parallel and Distributed Computing, Applications and Technologies.*
[29] 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.*