EasyChair Preprint

№ 7733

The Cost of Time Virtualization in Linux Containers

Xavier Merino and Carlos Otero

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

April 6, 2022

The Cost of Time Virtualization in Linux Containers

Xavier Merino
Dept. Computer Engineering and Sciences
Florida Institute of Technology
Melbourne, FL, USA
xmerino2012@my.fit.edu

Dr. Carlos E. Otero
Dept. Computer Engineering and Sciences
Florida Institute of Technology
Melbourne, FL, USA
cotero@fit.edu

Abstract— **With the advent of containerization, applications are subjected to a dynamic operating model that requires them to be deployed in ever-changing environments, restarted, updated, migrated, and even rolled back to previously working versions on a frequent basis. Because there are no strict guarantees on host placement or scheduling, an application may be restarted in a new host or at a later time, thereby losing their sense of time and refusing service owing to incongruent states. Until now, process time has been linked to a server. With the recent introduction of the Linux time namespace, it is now feasible to link time to a service. Processes under a time namespace can obtain a timeline of their own, irrespective of the host. Support for the time namespace is currently lacking in the most popular container engines. In this work, we present a workflow for building containers that leverage the time namespace, as well as the first analysis on the performance cost incurred by virtualizing time in Linux containers using this namespace. We consider 11 time-related system calls and their vDSO variants, making this one of the most comprehensive studies on the overhead of time virtualization in the literature.**

Keywords—virtual time, containers, microservices, virtualization, namespaces

I. Introduction

Virtual time is defined as a technique for organizing distributed systems using a temporal coordinate system that is more computationally relevant than real time since it supports mechanisms for synchronization and concurrency control [1]. Virtual time does not have to be connected to real time; in fact, it is possible to have numerous local virtual clocks that are only weakly synced and all progressing toward higher virtual times. They may occasionally jump backwards, but the general trend is for them to be monotonic. When it comes to virtual time, multiprocessing systems and distributed systems have some similarities [2], and each process and action can be described by the temporal coordinates (x, t), where x is the process and t is the instant in its virtual time. In the absence of virtual time, all processes share the host's chronology.

To a limited extent, all Linux processes are already under virtual time. After all, the Linux Completely Fair Scheduler (CFS) introduces the concept of virtual runtime [3] (i.e., a measure of the runtime of a thread) for scheduling purposes [4]. Because the aim of CFS' virtual time is to improve the performance of interactive processes in desktop systems [5], it is not intended to reach beyond the confines of a host. Today, however, microservices are designed to be loosely coupled and scalable, with the ability to make use of disposable resources in dynamically provisioned settings. As a result, applications are typically divided into smaller components that are frequently restarted, updated [6], migrated [7], and even rolled back to previously well-known versions [8]. This modern operating model requires processes to be dynamically scheduled and there are no guarantees that a process, once halted, will resume in the same host or within a specified time frame. At migration time, a new process is created on the receiving host, and CFS's virtual time notion is insufficient to prevent the process from believing that it has advanced or regressed in time. This may lead to service refusal due to an incongruent perception of system time [9] or a network timeout. This is because the concept of time is linked to a server rather than a service.

The concept of virtualizing time for each process is not new. One of the earlier attempts to merge some code into the Linux kernel allowed a process to keep its own view of time by keeping offsets that were added to the current system time [10]. At the time, the only use case was to speed up the `gettimeofday()` system call by keeping a copy of time in userspace. However, the introduction of the vDSO [11], a kernel read-only shared library mapped into all userspace applications, became the standard for accelerated system calls and this attempt at virtual time did not make it to the kernel. Over the years, with the emphasis on OS-level virtualization techniques that had become popular as a result of containers, the Linux maintainers added support for a time virtualization approach, the time namespace [12]. This enables an application to maintain its own sense of time using the monotonic and boot time clocks. Unfortunately, outside of research environments, the usage of this namespace in microservices is not viable because it is not yet included in the Open Container Initiative (OCI) specification [13] and thus is not supported by mainstream container engines [14] [15] and orchestrators. Furthermore, due to implementation details, this namespace can only virtualize time reporting but not the rate of passage of time [16], limiting the full potential of virtual time.

Because of the recent introduction of time namespaces, support in container engines is non-existent and very little is known about the impact of virtual time on an application's performance. Furthermore, there is no published work on the performance implications of this new namespace. In this work, we present a workflow for manually creating containers that leverage this new feature and present the first assessment of the overhead of time virtualization in Linux containers. We cover a wide variety of time-related system calls, and their vDSO counterparts, that may be impacted by the namespace, including those that are related to sleep, time reporting, and process timers. More specifically, we assess the overhead associated with using

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

time virtualization in two scenarios: (1) when it is used independently from other container-enabling primitives, and (2) when all the typical characteristics of a production-grade container are applied. We compare these findings to a baseline of no-time virtualization in which the host provides time information unmodified. We search the literature for prior virtual time implementations in OS-level virtualization environments in order to draw comparisons. To the best of our knowledge, this is the most comprehensive analysis of the impact of time virtualization on runtime overhead so far.

This work is divided into sections. Section 2 provides a brief description of timekeeping in Linux and the POSIX clocks that are available as of kernel version 5.13, as well as the ones which are virtualized using the newly introduced time namespace. We also discuss the role that namespaces play in supplying the primitives required to enable the segregation of processes via their own worldview, hence allowing the creation of containers. Section 3 provides a workflow for the manual building of containers that leverage the time namespace. Section 4 presents the system calls examined and describes the process used to collect data on the performance overhead incurred by using time namespaces at different virtualization levels. Section 5 describes the methodology used to compare the different virtualization scenarios. Section 6 discusses the comparison results. Section 7 presents other implementations of virtual time in the literature and, when available, their performance overhead. Section 8 presents concluding remarks and future avenues of research.

II. Background

In this section we present a brief introduction of Linux's timekeeping mechanisms and the clocks available for application use. We also provide an overview of containers and the Linux primitives that enable them, highlighting the recently introduced time namespace.

A. Linux Timekeeping

A variety of timekeeping devices (e.g., TSC, PIT, RTC, ACPI, HPET, and LAPIC) are available in modern hardware and can serve as clock sources for a system [17]. The purpose of a clock source is to give a chronology for the system, indicating where you are in time. In general, a system needs an adequate clock source that never goes backward, never stops ticking, avoids discontinuous time jumps, has a reasonable resolution or frequency, and is easily accessible to userspace code. In practice, this means that a clock source must be monotonic (i.e., always increasing), have high precision and a constant frequency, not move suddenly in time, and offer atomic access regardless of the underlying hardware design. The TSC (Time Stamp Counter), an auto-incremented CPU register, is the preferred clock source on current x86 hardware because the remaining timers do not meet the criteria or are not always accessible [18]. This is not to say that TSC is without flaws; rather, the flaws revealed by TSC are an acceptable compromise when contrasted to the alternatives.

Timekeeping in Linux is accomplished via the concept of clocks, which are abstractions on the previously described hardware counters. Rather than making use of the clock sources directly (e.g., via the `rdtsc` instruction for the TSC) [19], applications can interface with Linux clocks for timekeeping. This allows for faster access than might be available for a clock source. For instance, the Completely Fair Scheduler (CFS) needs timing information very often, and while the clock sources might be very accurate, access to those is not fast enough. In such cases, sacrificing accuracy for speed is required, as accomplished in the CFS via the `sched_clock()` kernel function [20]. In other userspace scenarios, such as high-performance databases and financial applications, the latency incurred by accessing the clock source may be excessive, necessitating the compromise of using Linux clocks. Linux has multiple clock variants and has added several options over time to accommodate the needs of applications and speed up the retrieval of information [21]. Table 1 displays a list of Linux clocks and their features.

Applications may access clock related functionality via system calls. This may include obtaining the current date and time, the resolution of a clock, and setting interval timers on top of existing clocks. Because system calls are expensive due to the mode switch from user mode to kernel mode, several optimizations, such as the use of the vDSO (virtual dynamic shared object) have been implemented, particularly for frequently used system calls relating to time. The vDSO is a small, shared library that is mapped into the address space of all userspace applications to make kernel information available fast and reduce the calling overhead. See Table 2 for a list of time-related system calls that we explore in this work.

Nowadays, virtualization has become widespread. Hypervisor-based virtualization adds complications to timekeeping. For example, during virtual machine migration, the TSC value may experience a jump and the frequency may vary between hosts. As a result, numerous initiatives to improve the TSC have been recorded , including paravirtualized clocks (e.g., pvclock, kvmclock, TSC page) [22] and hardware additions such as TSC scaling. In OS-level virtualization, the

TABLE I. Available Clocks in Linux

Clock	Function	Introduced	Affected by Adjustments
`CLOCK_REALTIME`	Reports the number of seconds since the Epoch.	Linux 2.6	Yes
`CLOCK_MONOTONIC`	Reports number of seconds since kernel booted.	Linux 2.6	Partially (only adjtime and NTP)
`CLOCK_PROCESS_CPUTIME_ID`	Measures CPU time consumed by all threads of a process.	Linux 2.6.12	No
`CLOCK_THREAD_CPUTIME_ID`	Measures CPU time consumed by a thread.	Linux 2.6.12	No
`CLOCK_MONOTONIC_RAW`	Like CLOCK_MONOTONIC but without adjustments.	Linux 2.6.28	No
`CLOCK_REALTIME_COARSE`	Faster but less precise than CLOCK_REALTIME.	Linux 2.6.32	Yes
`CLOCK_MONOTONIC_COARSE`	Like CLOCK_MONOTONIC but less precise.	Linux 2.6.32	Partially (only adjtime and NTP)
`CLOCK_BOOTTIME`	Like CLOCK_MONOTONIC but counts suspended time.	Linux 2.6.39	Partially (only adjtime and NTP)
`CLOCK_REALTIME_ALARM`	Interval timer on CLOCK_REALTIME with system waking.	Linux 3.0	Yes
`CLOCK_BOOTTIME_ALARM`	Interval timer on CLOCK_BOOTTIME with system waking.	Linux 3.0	Partially (only adjtime and NTP)
`CLOCK_TAI`	Reports International Atomic Time ignoring leap seconds.	Linux 3.10	No

TABLE II. TIME-RELATED SYSTEM CALLS

System Call	Description	vDSO (x64)
`clock_getres()`	Finds resolution of the given clock.	No
`clock_gettime()`	Retrieves the time of the specified clock.	Yes
`gettimeofday()`	Gives the time since the Epoch in seconds and microseconds.	Yes
`nanosleep()`	Suspends execution until time has elapsed or process is killed.	No
`clock_nanosleep()`	Suspends execution until time has elapsed in specified clock or process is killed.	No
`timer_create()`	Creates a per-process interval timer on the specified clock.	No
`timer_settime()`	Starts or stops a specified timer.	No
`timer_getoverrun()`	Gets specified timer's overrun count.	No
`timer_delete()`	Stops and delete a specified timer.	No
`timerfd_create()`	Creates a timer on a specified clock and returns a file descriptor.	No
`timerfd_settime()`	Starts or stops a file descriptor-based timer.	No

host OS provides the reference for time for all containerized loads. Migration of containers is also prone to suffering discontinuous jumps in time [9].

While this has been addressed for virtual machines, it is still a challenge in the widely adopted container engines. The introduction of the time namespace is an attempt to give each container its own view of time, allowing it to preserve its time perception even after being migrated.

B. Container & Namespaces

Containers are a type of OS-level virtualization that allows developers to package an application into an image. Containers are portable and consistent throughout the development pipeline because they encapsulate all application needs, allowing for a smoother transition from development to testing and production environments [23]. Furthermore, containerized processes are segregated from the rest of the system, allowing an application to have its own worldview no matter where it runs. Containers have grown in popularity and are commonly utilized nowadays because of their dependability, scalability, and flexibility [24].

A container, unlike a Jail [25] in BSD or a Zone [26] in Solaris, is not a first-class concept in Linux [27]; instead, containers are built up of a combination of Linux primitives such as namespaces and control groups (cgroups). A namespace is a feature of the Linux kernel that allows an application to have its own view of a resource that is separate from the global resources [28]. When paired with cgroups, which allow monitoring and limiting resource utilization, a container can be limited to using a specific portion of its available resources (e.g., CPU, RAM, PIDs, RDMA, block I/O, etc.) [29]. This makes containers highly flexible and sophisticated, but it also allows for scenarios that would not be conceivable with Jails and Zones. For example, an application can be deployed in a container that is isolated from the rest of the system while sharing only the network namespace with another container for network traffic inspection. This type of resource sharing has become widespread due to the use of container engines and orchestrators (e.g., Kubernetes, Nomad, Mesos, etc.) and demonstrates container flexibility.

Traditionally, Linux supported namespacing (i.e., isolating) the control group root directory, the IPC and POSIX message queues, network devices and stacks, mount points, process IDs (PIDs), user and group IDs, and hostnames. Recently, as of kernel version 5.6, it is possible to isolate boot and monotonic clocks through the time namespace, giving an application its own view of time irrespective of the host. In other words, this enables the use of virtual time in containerized applications.

III. REFERENCE IMPLEMENTATION

In this section, we present a reference implementation for using the time namespace in containers. Typically, a container engine (e.g., Docker, podman) would be used to deploy a container with the default presets. The container engine would make certain that the appropriate namespaces and control groups were applied to the application. However, support for the newly proposed time namespace is not yet implemented in popular container engines, necessitating manual application deployment. We demonstrate the building of a standard container from scratch using the `util-linux` package's `unshare` command [30]. Alternatively, you may achieve the same result by using the `unshare()` system call [31] with the appropriate flags (e.g., `CLONE_NEWNET`, `CLONE_NEWNS`, etc.).

Fig. 1 depicts an activity diagram describing the procedure required to run an application under a time namespace as responsibility moves from the host to the newly constructed container. While this diagram only shows the configuration needed for the time namespace, it is expected that the other namespaces have been correctly configured as additional options to the unshare command. We also assume that the `/proc` pseudo-filesystem is mounted in the container.

The process on the host begins with the download of a base image from an image repository (#1). This is similar to how

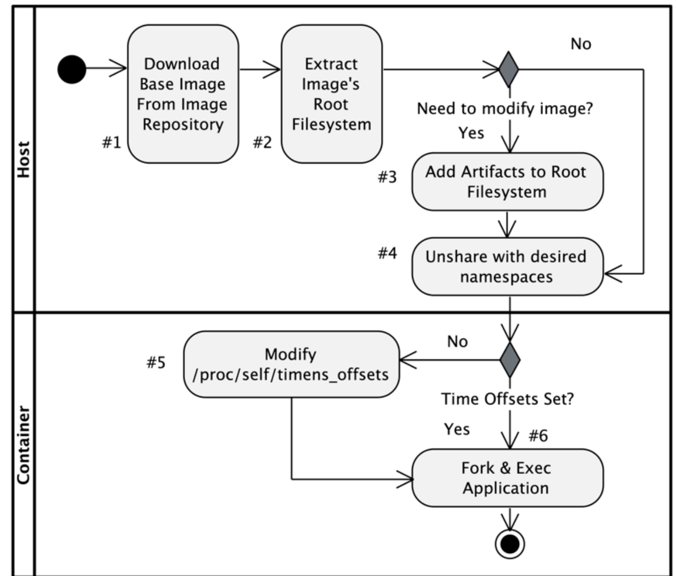

Fig. 1. Workflow to Implement Time Namespaces

Docker uses base images in a Dockerfile with the FROM command. The image's root filesystem is then extracted (#2). This is possible with `docker export` or `podman`. If the root filesystem lacks all of the artifacts required to deploy an application, you should modify it to incorporate those files (#3). It's worth noting that, unlike Docker, we don't employ a layered filesystem (e.g., AUFS, OverlayFS) [32], so all modifications to the root filesystem are persistent. When the root filesystem is complete, run the `unshare` command with the proper flags to utilize the desired namespaces (#4). In addition, similar to how `chroot` [33] works, you must select a change of root to the root filesystem directory. The namespaces are created by the `unshare` command (and hence applies the concept of a container). If the time offsets for the `CLOCK_BOOTTIME` and `CLOCK_MONOTONIC` clocks were not supplied when executing the `unshare` command, you will need to change the container's `/proc/self/timens_offsets` file with the desired offsets (#5). This must be completed before running any application. After you've established the offsets, you must fork and use exec to run your application (#6). This can be avoided if the fork flag is provided to the `unshare` command. The application will then run in the specified namespaces, with its own view of time. We have applied no resource constraints in our implementation, but we have enabled the use of the mount, UTS, user, IPC, network, PID, cgroup, and time namespaces. We used the official Ubuntu 21.10 image as the foundation for our container and added binaries to the `/usr/bin` directory to incorporate other applications.

IV. Data Collection

We aim to evaluate the performance impact of time virtualization on applications when compared to a non-virtualized baseline. We collect execution timing information on 11 time-related system calls and, when available, their vDSO-accelerated counterparts. In this study, we refer to the collection of system call data as metrics. The metric name is derived from the system call name, and when we specify the clock or access mode, we label it accordingly (i.e., `vdso_gettimeofday`, `clock_gettime_boottime`). Table 2 contains a list of the system calls that we analyze, their intended use, and whether they have a vDSO equivalent. Table 3 contains a list of the metrics for which we collect execution time (in nanoseconds).

We use `uftrace` [34] to acquire timing information about system calls. `uftrace` is a tool primarily inspired by `ftrace` that allows the tracing of C/C++ applications built with compiler instrumentation. `uftrace` is a robust tool for tracing user space functions, library functions, Linux kernel functions, and system events. We chose `uftrace` over alternatives like `strace` and `ltrace` because it combines the features of those two tools while having a lower tracing overhead than both.

Based on the extent of virtualization used, we explored three levels:

1) Non-virtualized: The application is deployed directly on the host, sharing all accessible global resources. There are no resource constraints, and the application runs as root. This is referred to as the baseline level.

TABLE III. Metrics Collected

Metric	Description
`clock_getres`	Finds resolution of `CLOCK_MONOTONIC`.
`clock_gettime_monotonic`	Retrieves the time of `CLOCK_MONOTONIC`.
`vdso_clock_gettime_monotonic`	Retrieves the time of `CLOCK_MONOTONIC` using the vDSO.
`clock_gettime_boottime`	Retrieves the time of `CLOCK_BOOTTIME`.
`vdso_clock_gettime_boottime`	Retrieves the time of `CLOCK_BOOTTIME` using the vDSO.
`gettimeofday`	Retrieves the time of `CLOCK_REALTIME`.
`vdso_gettimeofday`	Retrieves the time of `CLOCK_REALTIME` using the vDSO.
`nanosleep`	Suspends execution until time has elapsed in `CLOCK_REALTIME`.
`clock_nanosleep`	Suspends execution until time has elapsed in `CLOCK_MONOTONIC`.
`timer_create`	Creates a per-process interval timer on `CLOCK_MONOTONIC`.
`timer_settime`	Starts or stops a timer on `CLOCK_MONOTONIC`.
`timer_getoverrun`	Gets timer's overrun count (on `CLOCK_MONOTONIC`).
`timer_delete`	Stops and delete a specified timer (on `CLOCK_MONOTONIC`).
`timerfd_create`	Creates a timer on `CLOCK_MONOTONIC` and returns a file descriptor.
`timerfd_settime`	Starts or stops a file descriptor-based timer (on `CLOCK_MONOTONIC`).

2) Time Namespace Only: The application is deployed directly on the host, sharing all global resources except the system's perception of time. This is accomplished via the time namespace by setting offsets for the `CLOCK_BOOTTIME` and `CLOCK_MONOTONIC` clocks. There are no resource constraints, and the application runs as root. This is referred to as the namespace level.

3) Container: The application is deployed in a hand-built container and is isolated by using the eight available namespaces (i.e., cgroup, IPC, network, mount, PID, time, user, UTS). Namespaces provide the container with an isolated instance of the global resource, limiting what is shared. The container uses Ubuntu 21.10 as its base image. The time namespace was set up in the same way as the previous namespace scenario, by providing offsets for the `CLOCK_BOOTTIME` and `CLOCK_MONOTONIC` clocks. There are no resource constraints, and the application runs as root. This is referred to as the container level.

We performed the tests in a virtual machine instance (4 vCPUs, 8 GB RAM) hosted in a 12-core AMD Ryzen 9 3900X. The hypervisor used in the test was Oracle's VirtualBox. Both the virtual machine and the host OS were running Ubuntu 21.10 with kernel version 5.13.0-22-generic (x86_64). We made sure that both the containers and the host OS have the same kernel version as to avoid any performance issues that could be caused by "mismatched" kernels [35]. We preferred testing locally over testing in AWS instances because, although the latest Amazon Linux AMI contains kernel version 5.10 (which supports time namespaces), the `util-linux` packages is outdated (version 2.30.2) and the `unshare` command does not recognize the time

namespace as a valid option. We also chose the vCPU and RAM configuration to mimic the resource allocation that AWS provides to a xlarge, compute-optimized, instance. In terms of software, the VM made use of Ubuntu's `glibc` 2.34-0ubuntu3, `gcc` 11.2.0, `util-linux` 2.36.1, and `uftrace` v0.9.4. The host OS had Oracle's VirtualBox 6.1.26r145957 installed as the hypervisor. VirtualBox maintains synchronization of all guest-visible time sources with the monotonic host time [36].

For each system call, we collect 200 records, with each record representing the average of 1 million system call executions. This is done to obtain a more representative sample and to amortize the cost of the initial vDSO call, which normally results in a page fault [37].

V. Comparison Methodology

We used hypothesis testing to determine whether time virtualization (with varying degrees of OS-level virtualization) degrades system performance by increasing the execution time of time-related system calls as compared to the baseline group. We divided the tests into three groups based on the extent of OS-level virtualization used: non-virtualized (n = 200, baseline), namespace (n = 200), and container (n = 200). We assess normality using the Shapiro-Wilk's test and confirm suspicions of violation of normality by visual inspection of Q-Q plots. Although not ideal from a statistical standpoint, we maintained the outliers because there is no reason to reject them or declare them invalid. We assess homogeneity of variances via Levene's test.

A one-way Welch ANOVA was conducted for each metric that exhibited heteroscedasticity to determine if the execution time differences between groups with varied levels of OS virtualization were statistically significantly different. For those metrics that met the assumption of homogeneity of variances, a one-way ANOVA was conducted instead. We express the difference in group means as a null H_0 (all group population means are equal) and alternative H_A hypothesis (the means of the groups are not equal). These tests are appropriate due to its robustness against deviation of normality, particularly when sample sizes are large and equal. For the cases where there is evidence that the null hypothesis can be rejected (i.e., there is a statistically significant difference between the means of the groups), we conduct a post-hoc test (i.e., Tukey HSD for ANOVA, Games-Howell for Welch's ANOVA) to determine where the differences lie. Data is presented as mean ± standard deviation. In all cases, and for all tests, we report results at the 95% confidence level.

VI. Results And Discussion

We analyze the collected data and present the results in this section. To better discuss our findings, we divide the system calls into three groups depending on their functionality: sleep (which is concerned with system calls that cause delays), time (which is concerned with system calls that report time), and timers (concerned with system calls that arm and disarm per-process timers). Table 4 summarizes our findings; refer to the table for more information on the statistical significance of the tests, the difference between the virtualization levels' means, and their confidence intervals.

A. *Sleep Inducing System Calls*

We included `clock_nanosleep()` and `nanosleep()` in the group of system calls that cause delays. Both system calls implement high-resolution sleep (i.e., suspending the calling thread's execution) until the specified time expires, a signal triggers a handler in the calling thread, or the process is terminated [38] [39]. If the requested time is not an exact multiple of the clock's granularity, the interval will be rounded up to the next multiple. We aimed at a 100-millisecond delay per system call in our experiments. The metrics associated with this group of system calls have the same name as the system calls from which they were produced.

Unlike `clock_nanosleep()`, which counts time against a user-specified clock, `nanosleep()` uses the `CLOCK_REALTIME` clock in POSIX.1 conforming implementations. In Linux, `CLOCK_MONOTONIC` is used for `nanosleep()` since it is unaffected by discontinuous jumps in the system time, an aspect which is required for `nanosleep()` when using `CLOCK_REALTIME`. As a result, Linux's version is functionally equivalent to the POSIX-compliant implementation. Another distinction between the two is that `clock_nanosleep()` can sleep until an absolute time has been met, rather than relying on sleep intervals alone. This provides the user with additional control and stops a process from sleeping for an extended period if the time has been adjusted by an administrator or updated via NTP.

The execution time of `clock_nanosleep()` rose from the baseline (101.57ms ± 5.61ms) to the namespace (102.19ms ± 6.26ms), to the container level (102.29ms ± 6ms). Similarly, the execution time of `nanosleep()` increased from the baseline (101.12ms ± 2.36ms) to the namespace (101.34ms ± 2.38ms), to the container level (101.59ms ± 3.07ms), in that order. The differences between the virtualization levels in both system calls were not statistically significant.

Because the sleep mechanism is mostly dependent on the scheduler, it was expected that there would be little difference in sleeping calls across virtualization levels. A sleeping task surrenders the processor to the scheduler, exposing the sleeping job to additional scheduling latency, which may result in the sleeping task not executing immediately after the sleeping interval ends. Additionally, because the kernel may be executing other sleeping tasks while the process sleeps, additional sources of latency may be introduced. As other processes execute, the tasks may also compete for the instruction cache, potentially introducing stalls due to cache misses when the process resumes. While we haven't quantified it, and it's possible to do so using performance counters, we attempted to limit this effect as much as possible by running the sleeping tests exclusively. It is critical to note that Linux is not a real-time operating system and hence cannot guarantee precise sleep intervals. However, you can rely on sleeping for an interval close to the specified one [40] because all processes, including containerized ones, are scheduled by the same kernel. As a result, Madden [41] advocated that when monopolizing the processor is not an issue, the use of `nanosleep()` be discouraged in favor of busy wait. Notably, when the Completely Fair Scheduler (CFS) was made the default scheduler, the granularity of sleeping system calls improved [42].

TABLE IV. SUMMARY OF FINDINGS

Metric	Descriptives			ANOVA				Post Hoc Tests					
	Level	Mean (ns)	Std. Dev (ns)	df1	df2	F	Sig.	(I) Level	(J) Level	Mean Diff. (I-J)	Sig.	95% CI LB	95% CI UB
clock_ nanosleep	baseline	1.02E+08	5.61E+06	2	597	0.85	0.426	N/A					
	namespace	1.02E+08	6.26E+06										
	container	1.02E+08	6.00E+06										
nanosleep[a]	baseline	1.01E+08	2.36E+06	2	393	1.48[b]	0.229	N/A					
	namespace	1.01E+08	2.38E+06										
	container	1.02E+08	3.07E+06										
clock_ getres[a]	baseline	129.92	2.594	2	392	391.54[b]	<0.001	namespace	baseline	3.440[c]	<0.001	2.85	4.03
	namespace	133.36	2.383						container	0.405	0.150	-0.11	0.92
	container	132.96	1.934					container	baseline	3.035[c]	<0.001	2.50	3.57
clock_ gettime_ boottime	baseline	146.85	1.913	2	597	26.00	<0.001	namespace	baseline	1.215[c]	<0.001	0.77	1.66
	namespace	148.06	2.004						container	0.070	0.927	-0.37	0.51
	container	147.99	1.751					container	baseline	1.145[c]	<0.001	0.70	1.59
clock_ gettime_ monotonic[a]	baseline	146.09	1.363	2	389	389.29[b]	<0.001	namespace	baseline	1.205[c]	<0.001	0.80	1.61
	namespace	147.29	1.999						container	0.380	0.091	-0.05	0.81
	container	146.91	1.598					container	baseline	.825[c]	<0.001	0.48	1.17
gettimeofday[a]	baseline	121.37	1.296	2	358	357.90[b]	<0.001	namespace	baseline	2.610[c]	<0.001	2.09	3.13
	namespace	123.98	2.833						container	0.535	0.091	-0.06	1.13
	container	123.44	2.221					container	baseline	2.075[c]	<0.001	1.65	2.50
timer_ create[a]	baseline	282.56	5.219	2	396	395.95[b]	<0.001	namespace	baseline	3.265[c]	<0.001	1.94	4.59
	namespace	285.83	5.980						container	-1.285	0.086	-2.71	0.14
	container	287.11	6.097					container	baseline	4.550[c]	<0.001	3.21	5.89
timer_ settime[a]	baseline	192.23	4.223	2	396	396.08[b]	<0.001	namespace	baseline	1.805[c]	<0.001	0.85	2.76
	namespace	194.03	3.666						container	-0.780	0.132	-1.73	0.17
	container	194.81	4.223					container	baseline	2.585[c]	<0.001	1.63	3.54
timer_ delete[a]	baseline	227.05	3.822	2	395	395.17[b]	<0.001	namespace	baseline	2.620[c]	<0.001	1.67	3.57
	namespace	229.67	4.226						container	-0.925	0.098	-1.98	0.13
	container	230.59	4.711					container	baseline	3.545[c]	<0.001	2.54	4.55
timer_ getoverrun[a]	baseline	172.18	7.403	2	391	391.33[b]	<0.001	namespace	baseline	2.795[c]	<0.001	1.25	4.34
	namespace	174.97	5.636						container	-3.190[c]	<0.001	-4.71	-1.67
	container	178.16	7.200					container	baseline	5.985[c]	<0.001	4.27	7.70
timerfd_ create[a]	baseline	576.83	8.572	2	393	392.70[b]	<0.001	namespace	baseline	12.255[c]	<0.001	10.06	14.45
	namespace	589.08	10.017						container	-0.390	0.930	-2.91	2.13
	container	589.47	11.378					container	baseline	12.645[c]	<0.001	10.27	15.02
timerfd_ settime	baseline	241.17	3.456	2	597	203.84	<0.001	namespace	baseline	8.280[c]	<0.001	7.21	9.35
	namespace	249.45	5.407						container	-0.335	0.809	-1.60	0.93
	container	249.79	5.378					container	baseline	8.615[c]	<0.001	7.55	9.68
vdso_clock_ gettime_ boottime[a]	baseline	56.81	1.091	2	396	396.44[b]	<0.001	namespace	baseline	3.075[c]	<0.001	2.83	3.32
	namespace	59.89	0.947						container	-.450[c]	<0.001	-0.69	-0.21
	container	60.34	1.058					container	baseline	3.525[c]	<0.001	3.27	3.78
vdso_clock_ gettime_ monotonic[a]	baseline	56.39	0.769	2	386	386.28[b]	<0.001	namespace	baseline	3.565[c]	<0.001	3.37	3.76
	namespace	59.96	0.920						container	-.405[c]	0.001	-0.66	-0.15
	container	60.36	1.212					container	baseline	3.970[c]	<0.001	3.73	4.21
vdso_clock_ gettimeofday[a]	baseline	55.36	0.886	2	389	388.98[b]	<0.001	namespace	baseline	2.215[c]	<0.001	1.98	2.45
	namespace	57.58	1.114						container	-.560[c]	<0.001	-0.84	-0.28
	container	58.14	1.267					container	baseline	2.775[c]	<0.001	2.52	3.03

a. Welch's ANOVA used because metric exhibited heteroscedasticity.

b. Asymptotically F distributed.

c. The mean difference is significant at the 0.05 level.

B. Time Reporting System Calls

In the category of time reporting system calls, we included `clock_getres()`, `clock_gettime()`, `gettimeofday()`, and their vDSO counterparts. This is because these system calls report time or its precision. The system calls `clock_gettime()` and `gettimeofday()` obtain time information, whereas `clock_getres()` determines a clock's resolution. Unlike `clock_gettime()`, which allows you to choose the clock whose time you want to report, `gettimeofday()` reports on `CLOCK_REALTIME` only [43]. The following metrics are associated with this group: `clock_getres`, `clock_gettime_monotonic`, `clock_gettime_boottime`, `gettimeofday`, `vdso_clock_gettime_monotonic`, `vdso_clock_gettime_boottime`, and `vdso_clock_gettimeofday`. The execution time of all the metrics in this group was statistically significantly different amongst virtualization levels.

The execution time of clock_getres() rose from the baseline (129.92ns ± 2.594ns) to the namespace (133.36ns ± 2.383ns) level. From the namespace to the container level (132.96ns ± 1.934ns), there was a minor reduction. The execution time of the gettimeofday() rose from the baseline (121.37ns ± 1.296ns) to the namespace (123.98ns ± 2.833ns) level. From the namespace to the container (123.44ns ± 2.221ns) level, there was a slight decline. The execution time of clock_gettime() when CLOCK_BOOTTIME was specified increased from the baseline (146.85ns ± 1.913ns) to the namespace (148.06ns ± 2.004ns) level. There was a slight drop from the namespace to the container (147.99ns ± 1.751ns) level. When CLOCK_MONOTONIC was specified, the execution time rose from the baseline (146.09ns ± 1.363ns) to the namespace (147.29ns ± 1.999ns) level. From the namespace to the container (146.91ns ± 1.598ns) level, there was a minor decline. In the aforementioned system calls, the rise from the baseline to the namespace level, as well as the increase from the baseline to the container level, were statistically significant, while the difference between the namespace and container levels was not.

When evaluating the vDSO equivalents, the execution time of the accelerated clock_gettime() when CLOCK_BOOTTIME was specified rose in the following order: baseline (56.81ns ± 1.091ns), namespace (59.89ns ± 0.947ns), and container level (60.34ns ± 1.058ns). When using CLOCK_MONOTONIC, the time rose in the following order: baseline (56.39ns ± 0.769ns), namespace (59.96ns ± 0.920ns), and container level (60.36ns ± 1.212ns). The accelerated gettimeofday() execution's time grew incrementally from the baseline (55.36ns ± 0.886ns) to the namespace (57.58ns ± 1.114ns), and the container level (58.14ns ± 1.267ns). All vDSO calls showed statistically significant increases from the baseline to the namespace level, as well as from the namespace to the container level.

The system calls analyzed made use of CLOCK_REALTIME, CLOCK_MONOTONIC, and CLOCK_BOOTTIME. Of those three, only CLOCK_MONOTONIC and CLOCK_BOOTTIME (and their variants) are virtualized through the time namespace. Every non-vDSO call has a statistically significant difference from the baseline to any OS-virtualization level, but not between execution in a time namespace and a fully namespaced container. Because CLOCK_REALTIME is shared with the host OS, we did not expect a major difference between the baseline and the other virtualization levels when using gettimeofday(). The increase, however, was not unreasonably expensive, with our worst-case scenario adding ~2.15% overhead, in line with the 3%-5% percent overhead that comes with the use of containers [44] [45] [46]. Surprisingly, adding further virtualization characteristics (i.e., more namespaces) on top of the time namespace did not result in a substantial change in system call execution time. This demonstrates that the time namespace can be added to current container standards because is within acceptable overhead margins. When it comes to vDSO accelerated calls, the difference from the baseline is significant at every level of OS-virtualization (i.e., from a time namespace only to a full container) and across virtualization levels. The mean difference is greater when accessing the CLOCK_MONOTONIC and CLOCK_BOOTTIME clocks, with differences ranging from 2.83-3.78ns and 3.37-4.21ns, respectively, than when accessing CLOCK_REALTIME, with a mean difference ranging from 1.98-3.03ns. This could be related to the extra calculations that the kernel must undertake in order to maintain the time offsets for the virtualized clocks. We don't conduct additional tests to verify that vDSO calls are faster than system calls because (1) it is out of the scope of this study and (2) this is generally well-accepted since it has been shown that system calls have a higher access latency than the vDSO calls. Because of the access latency, these minor differences noted with the vDSO calls may have been amortized in the system call results, resulting in non-statistically significant differences between the namespace and the container levels.

C. Timer System Calls

We included timer_create(), timer_settime(), timer_getoverrun(), timer_delete(), timerfd_create(), and timerfd_settime() under the category of per-process timers system calls. These system calls create, arm, disarm, delete, and report on a timer's overrun count. The fd variants allow setting up timers through file descriptors, which is a more event-loop friendly approach (i.e., allow the use of select(), poll(), and the epoll API) than the one provided by conventional timers. The associated metrics for this system call group carry the same name as the system calls from which they were derived from. All system calls in this group had a statistically significant difference in execution time between virtualization levels.

The execution time of timer_create() increased from the baseline (282.56ns ± 5.219ns) to the namespace (285.83ns ± 5.980ns), and the container level (287.11ns ± 6.097ns), in that order. With respect to timer_settime(), the execution time grew from the baseline (192.23ns ± 4.223ns) to the namespace (194.03ns ± 3.666ns), and the container level (194.81ns ± 4.223ns), in that order. For timer_delete(), the execution time increased from the baseline (227.05ns ± 3.822ns) to the namespace (229.67ns ± 4.226ns), and the container level (230.59ns ± 4.711ns), in that order. The fd variants exhibited a similar pattern. The execution time of the timerfd_create() increased from the baseline (576.83ns ± 8.572ns) to the namespace (589.08ns ± 10.017ns), and the container level (589.47ns ± 11.378ns), in that order. The execution time of the timerfd_settime() rose from the baseline (241.17ns ± 3.456ns) to the namespace (249.45ns ± 5.407ns), and the container level (249.79ns ± 5.378ns), in that order. The increase from the baseline to the namespace level, as well as the increase from the baseline to the container level, were statistically significant in the aforementioned system calls, while the difference between the namespace and container levels was not.

The execution time of the timer_getoverrun() system call climbed from the baseline (172.18ns ± 7.403ns) to the namespace (174.97ns ± 5.636ns), and the container level (178.16ns ± 7.200ns). The rise from the baseline to the

namespace level, as well as the increase from the namespace to the container level, were statistically significant.

The results reveal that there appears to be no discernible change in execution times for creating, arming, disarming, and deleting timers on `CLOCK_MONOTONIC`, in both conventional and file descriptor system call variants. The file descriptor variants, on the other hand, exhibit higher mean execution times, which is most likely owing to the underlying opened files interacting with the timer itself. The only exception was `timer_getoverrun()`, which had statistically significant differences in execution times across all virtualization levels. There is no overrun system call available for the file descriptor variants as this is obtained via `read()` on the file descriptor. The time to create a timer increased by 2.19%, the time to arm/disarm a timer climbed by 3.57%, and the time to delete a timer increased by 1.56% under our worst-case scenario. The overhead in all circumstances is less than 4%, which is comparable to the overhead associated with containerized loads.

VII. Related Works

Other attempts at virtual time have been documented in the literature. In our search, we excluded works that made exclusive use of virtual machines and favored those that used some form of OS-level virtualization (e.g., namespaces, containers, jails, zones). From those works, we cover their use of virtual time and any discussion of the performance overhead that the authors made available.

Zheng and Nicol [44] developed a virtual time system that simulated functional and temporal behavior of network communication by trapping the execution of system calls to return an illusion of virtual time as required by the simulation using OpenVZ's Virtual Environments. This is one of the earliest attempts at using OS-level virtualization to decouple the virtualization of execution and time to prevent the execution from reflecting the host's serialization of tasks. Their approach requires changes to the OpenVZ kernel so that tasks are correctly scheduled and modifications to system calls such as `gettimeofday()`. In their experiments they noticed an overhead of 4.9% over the non-virtualized baseline. Jin et al. [47] expanded the work and employed this custom kernel to support a parallel network simulator, S3F. They added the ability to advance in virtual time only when there is activity in an application or network. Although unmentioned, we assume that they inherit the same overhead from Zheng and Nicol's [44] work based on their adoption of the custom kernel.

In TimeKeeper, Lamps et al. [48] introduced a set of Linux kernel modifications to embed Linux Containers (LXC) into virtual time for network simulation. The main concept is to give each container a dilated view of time to make it seem as if time advances more slowly than real time to make network resources appear faster. To achieve this, they added to the Linux `task_struct` to include a time dilation factor (TDF) and other variables related to timekeeping, exposed an API to control time operations (e.g., dilation, freeze/unfreeze, time leaping, etc), modified system calls (e.g., `gettimeofday()`, `sleep()`, `poll()`), and used `hrtimers` to schedule the execution of containers. The choice of `hrtimers` ended up affecting the accuracy of virtual time, but in 90% of the cases they were able to keep virtual time within 4us of the expected virtual time. This is because their approach lacks flexibility in scheduling by subjecting a process to a fixed execution time slice. When evaluating their approach for overhead, they did not evaluate each modification for overhead but rather how many containers they could deploy while maintaining the validity of virtual time. They found that this was related to the time dilation factor used for the containers: $6/(TDF + 1)$.

TimeKeeper has been the inspiration for the saga of work presented by Yan and Jin. In [49] they expanded the fields added to `task_struct` and added a pair of system calls to unshare time and set the dilation factor. They applied those advancements to the work of Handigol et al. [50], to provide the emulator, Mininet-Hifi, with the ability of virtual time. They enhanced it by adding a freezer system capable of stopping and resuming a container's virtual time. This was a source of trouble for Mininet-Hifi since containers use the same system clock of the physical machine and this leads to a wrong perception of time because a container's clock keeps advancing even if it is not running. In [51] and [52] Yan and Jin reshaped their work into a Linux namespace, the clock namespace. The final implementation, VT-Mininet, resembles modern day practices because there is use of control groups (via Mininet-Hifi), container primitives, and traffic shaping. In their tests, the average overhead for a `gettimeofday()` system call was 13ns. Because some time related system calls are accelerated via vDSO, the containers are able to bypass the features implemented by Yan and Jin. As a workaround, they disabled the use of vDSO for those system calls. Other forks of Mininet include Containernet [53] where Docker containers are used instead of the manual combination of container-building primitives.

Navarro et al. [54] implemented virtual time by using a counter of time-related system calls invoked by a process. This ensures a monotonic logical time that can be used to guarantee reproducibility of execution stages in containerized processes. In their implementation, they intercept vDSO calls and replace them with system calls. We are only concerned with the overhead of their virtual time solution, but this is not explored. However, the higher access latency of system calls when compared to vDSO calls is widely documented.

Another use of OS-level virtualization is the use of Jails in BSD-based systems. Grau et al. [55] used a hybrid approach combining VMs, Jails, and BSD's Virtual Routing to provide time virtualization (based on time dilation) for virtual nodes (i.e., network processes). Their evaluation of the system did not include runtime overhead tests, rather, they focused on the memory consumption of each virtual node and discovered that their approach was lighter by consuming less memory than those implemented under Xen for Linux-based systems. Unfortunately, we can't directly compare their approach since no runtime overhead was given. Hibler et al. [56] modified Emulab to make use of FreeBSD Jails. Emulab uses the concept of virtual time to guarantee the ordering of events in an experiment [57]. The goal of the enhancements was to enable the emulation of systems larger than the underlying testbed through light virtualization. There was no mention of the overhead of virtualization as the authors considered that fidelity

was not as important for small scale simulations. Today, Emulab is able to use Docker containers as well as virtual machines [58]. In Solaris Zones, starting with Solaris 11, virtual time has been implemented by allowing each non-global zone to set its own time via the `clock_settime()` system call [59].

In comparison to the studies described above, our work delivers the most comprehensive study on the impact of virtual time on runtime overhead. Several time-related system calls were omitted. We did not discuss the system calls `time()`, `ftime()`, `sleep()`, and `clock_settime()`. We didn't include `time()` and `ftime()` because they were superseded by `clock_gettime()`. We only included `gettimeofday()` as a point of comparison with the research mentioned in this section as it has also been supplanted by `clock_gettime()`. The `sleep()` system call is not covered because it is implemented via `nanosleep()` [60]. We exclude `clock_settime()` because the time of a time namespace can only be set once, hence all applications that get forked after the initial setup inherit this configuration. It is worth noting that the official Linux namespace outperforms the alternatives and, unlike many other studies, can operate with accelerated vDSO calls.

VIII. Conclusions & Future Work

Virtualizing time is not a novel concept. It has been explored in the literature for hypervisor-based and OS-level virtualization systems. Recently, with the widespread adoption of containers, a new Linux primitive was added, the time namespace. This allows processes to "unshare" their perception of time from the host system. By decoupling their timelines, processes are able to keep their own perception of time, regardless of the host on which they execute. There is little information available about the performance overhead of time virtualization. In previous attempts, researchers focused primarily on the ability to dilate and virtualize time as a means of multiplexing resources, rather than as an intrinsic feature of a process. To meet the requirements of today's dynamic operating models, and as the microservices trend becomes stronger, the concept of time must be tied to a service rather than a server.

In this work we presented a workflow to leverage the newly introduced time namespace for creating containers that possess their own view of time. We subsequently tested 11 time-related system calls and their vDSO counterparts for runtime overhead when time virtualization was used. We evaluated the overhead associated with employing time virtualization in two scenarios: (1) when used independently from other container-enabling primitives, and (2) when all normal characteristics of a production-grade container are applied. We demonstrated that the runtime performance of a process running in a time-namespace differs little from that of a process running in a fully namespaced container. In our experiments, the performance impact was of less than 4% in the worst-case scenarios, with most system calls exhibiting an increase of ~2% in runtime. Because virtualizing time may have an impact on applications that rely heavily on timing mechanisms, it is critical to analyze the impact of virtual time in containers, particularly as they are becoming the typical approach of delivering applications in the cloud. According to our findings, applications running in virtual time will incur low overhead, comparable to that of currently employed OS-level virtualization techniques.

In the future, we intend to add a storage overlay solution to our handmade containers so that we can carry out direct comparisons against Docker containers.

References

[1] D. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems,* vol. 7, no. 3, pp. 404-425, 1985.

[2] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," in *Concurrency: the Works of Leslie Lamport*, New York, NY, USA, Association for Computing Machinery, 2019, pp. 179-196.

[3] The Kernel Development Community, "CFS Scheduler," 2022. [Online]. Available: https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html#overview.

[4] C. S. Wong, I. K. T. Tan, R. D. Kumari, J. W. Lam and W. Fun, "Fairness and interactive performance of O(1) and CFS Linux kernel schedulers," in *International Symposium on Information Technology*, Kuala Lumpur, Malaysia, 2008.

[5] J. Bouron, S. Chevalley, B. Lepers, W. Zwaenepoel, R. Gouciem, J. Lawall, G. Muller and J. Sopena, "The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS," in *USENIX ATC '18*, 2018.

[6] B. Burns, B. Grant, D. Oppenheimer, E. Brewer and J. Wilkes, "Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade," *Queue,* vol. 14, no. 1, pp. 70-93, 2016.

[7] L. Ma, S. Yi and Q. Li, "Efficient service handoff across edge servers via docker container migration," in *SEC '17: Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, San Jose, CA, 2017.

[8] X. Merino, C. Otero, M. Ridley and D. Elliott, "Managed Containers: A Framework for Resilient Containerized Mission Critical Systems," in *IEEE 11th International Conference on Cloud Computing (CLOUD)*, San Francisco, CA, USA, 2018.

[9] D. Elliott, "Vanishing Connections: Application Resiliency through Cross-Network TCP Migration," Florida Institute of Technology, Melbourne, FL, 2018.

[10] J. Dike, "[RFC] PATCH 0/4 - Time virtualization," 13 4 2006. [Online]. Available: https://lkml.org/lkml/2006/4/13/172.

[11] M. Frysinger, "vdso(7)," 27 8 2021. [Online]. Available: https://man7.org/linux/man-pages/man7/vdso.7.html.

[12] M. Kerrisk, "time_namespaces(7) - Linux manual page," 27 08 2021. [Online]. Available: https://man7.org/linux/man-pages/man7/time_namespaces.7.html#:~:text=Time%20namespaces%20virtualize%20the%20values,unspecified%20point%20in%20the%20past%22..

[13] opencontainers, "Add support for time namespace #1062," [Online]. Available: https://github.com/opencontainers/runtime-spec/pull/1062.

[14] "Support time namespaces #2345," [Online]. Available: https://github.com/opencontainers/runc/issues/2345.

[15] "Adding time namespace to the containers #39163," [Online]. Available: https://github.com/moby/moby/issues/39163.

[16] "Introduce Time Namespace," 12 11 2019. [Online]. Available: https://lore.kernel.org/lkml/20191112012724.250792-3-dima@arista.com/t/.

[17] V. Kuznetsov, "An Introduction to Timekeeping in Linux VMs," 2017. [Online]. Available: https://opensource.com/article/17/6/timekeeping-linux-vms.

[18] VMware, Inc., "Timekeeping in VMware Virtual Machines," VMWare, Inc., 2008.

[19] Intel Corporation, "Using the RDTSC Instruction for Performance Monitoring," 1998. [Online]. Available: https://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf.

[20] The Kernel Development Community, "Clock sources, Clock events, sched_clock() and delay timers," 2022. [Online]. Available: https://www.kernel.org/doc/html/latest/timers/timekeeping.html.

[21] M. Kerrisk, A. Brouwer and N. Clifford, "clock_gettime(3) — Linux manual page," 11 04 2020. [Online]. Available: https://www.man7.org/linux/man-pages/man3/clock_gettime.3.html.

[22] Red Hat, Inc., "Timekeeping Virtualization for x86-Based Architectures," 2010. [Online]. Available: https://www.kernel.org/doc/html/latest/virt/kvm/timekeeping.html?highlight=clock.

[23] Microsoft, "What is a container?," 2022. [Online]. Available: https://azure.microsoft.com/en-us/overview/what-is-a-container/#overview.

[24] Google Cloud, "What are containers?," [Online]. Available: https://cloud.google.com/learn/what-are-containers.

[25] The FreeBSD Project, "Chapter 15. Jails," in *FreeBSD Handbook*.

[26] D. Price and A. Tucker, "Solaris Zones: Operating System Support for Consolidating Commercial Workloads," *LISA,* vol. 4, pp. 241-254, 2004.

[27] J. Frazelle, "Research for practice: security for the modern age," *Communications of the ACM,* vol. 62, no. 1, pp. 43-45, 2019.

[28] M. Kerrisk and E. W. Biederman, "namespaces(7) — Linux manual page," 27 08 2021. [Online]. Available: https://man7.org/linux/man-pages/man7/namespaces.7.html.

[29] M. Kerrisk and S. Hallyn, "cgroups(7) — Linux manual page," 27 08 2021. [Online]. Available: https://www.man7.org/linux/man-pages/man7/cgroups.7.html.

[30] M. Kerrisk, "unshare(1) — Linux manual page," 20 06 2021. [Online]. Available: https://www.man7.org/linux/man-pages/man1/unshare.1.html.

[31] M. Kerrisk and J. Desai, "unshare(2) — Linux manual page," 22 03 2021. [Online]. Available: https://www.man7.org/linux/man-pages/man2/unshare.2.html.

[32] Docker Inc., "Docker storage drivers," 2021. [Online]. Available: https://docs.docker.com/storage/storagedriver/select-storage-driver/.

[33] D. Eckhard, "chroot(2) — Linux manual page," 22 03 2021. [Online]. Available: https://man7.org/linux/man-pages/man2/chroot.2.html.

[34] N. Kim, "uftrace: A function (graph) tracer for C/C++ userpace programs," [Online]. Available: https://uftrace.github.io/slide/#1.

[35] G. Borello, "In search of 0xffffffffff600400: troubleshooting containers, system calls and performance," 16 1 2018. [Online]. Available: https://sysdig.es/blog/troubleshooting-containers/.

[36] Oracle Corporation, "9.11. Fine Tuning Timers and Time Synchronization," 2022. [Online]. Available: https://www.virtualbox.org/manual/ch09.html#fine-tune-timers.

[37] A. Vagin, "[PATCH 4/6] arm64/vdso: Handle faults on timens page," 2 6 2020. [Online]. Available: https://lkml.iu.edu/hypermail/linux/kernel/2006.0/02544.html.

[38] M. Kerrisk, "clock_nanosleep(2) — Linux manual page," 22 03 2021. [Online]. Available: https://man7.org/linux/man-pages/man2/clock_nanosleep.2.html.

[39] M. Kerrisk and M. Kuhn, "nanosleep(2) — Linux manual page," 22 03 2021. [Online]. Available: https://man7.org/linux/man-pages/man2/nanosleep.2.html.

[40] M. Sollfrank, F. Loch, S. Denteneer and B. Vogel-Heuser, "Evaluating Docker for Lightweight Virtualization of Distributed and Time-Sensitive Applications in Industrial Automation," *IEEE Transactions on Industrial Informatics,* vol. 17, no. 5, pp. 3566 - 3576, 2020.

[41] M. M. Madden, "Challenges Using Linux as a Real-Time Operating System," NASA Langley Research Center, Hampton, VA, 2020.

[42] Red Hat, Inc., "usleep(3) and nanosleep(2) have better granularity with CFS scheduler in RHEL6," 14 October 2015. [Online]. Available: https://access.redhat.com/solutions/186663.

[43] D. Eckhard, "gettimeofday(2) — Linux manual page," 22 03 2021. [Online]. Available: https://www.man7.org/linux/man-pages/man2/gettimeofday.2.html.

[44] Y. Zheng and D. M. Nicol, "A Virtual Time System for OpenVZ-Based Network Emulators," in *2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*, Nice, France, 2011.

[45] W. Felter, A. Ferreira, R. Rajamony and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Philadelphia, PA, USA, 2015.

[46] A. Torrez, T. Randles and R. Priedhorsky, "HPC Container Runtimes have Minimal or No Performance Impact," in *IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, Denver, CO, USA, 2019.

[47] D. Jin, Y. Zheng, H. Zhu, D. M. Nicol and L. Winterrowd, "Virtual Time Integration of Emulation and Parallel Simulation," in *SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, Zhangjiajie, China, 2012.

[48] J. Lamps, D. M. Nicol and M. Caesar, "TimeKeeper: a lightweight virtual time system for linux," in *Proceedings of the 2nd ACM SIGSIM/PADS conference on Principles of advanced discrete simulation - SIGSIM-PADS '14*, Denver, Colorado, USA, 2014.

[49] J. Yan and J. Dong, "A lightweight container-based virtual time system for software-defined network emulation," *Journal of Simulation,* vol. 11, no. 3, pp. 253-266, 2017.

[50] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz and N. McKeown, "Reproducible network experiments using container-based emulation," in *8th international conference on Emerging networking experiments and technologies - CoNEXT '12*, Nice, France, 2012.

[51] J. Yan and D. Jin, "A Virtual Time System for Linux-container-based Emulation of Software-defined Networks," in *SIGSIM-PADS '15: SIGSIM Principles of Advanced Discrete Simulation*, London, UK, 2015.

[52] J. Yan and D. Jin, "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*, Santa Clara, CA, 2015.

[53] M. Peuster, H. Karl and S. van Rossem, "MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments," in *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Palo Alto, CA, USA, 2016.

[54] O. S. Navarro Leija, K. Shiptoski, R. G. Scott, B. Wang, N. Renner, R. R. Newton and J. Devietti, "Reproducible Containers," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems*, Lausanne, Switzerland, 2020.

[55] A. Grau, S. Maier, K. Herrmann and K. Rothernel, "Time Jails: A Hybrid Approach to Scalable Network Emulation," in *22nd Workshop on Principles of Advanced and Distributed Simulation*, Roma, Italy, 2008.

[56] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb and J. Lepreau, "Large-scale Virtualization in the Emulab Network Testbed," in *2008 USENIX Annual Technical Conference (USENIX ATC 08)*, 2008.

[57] C. Siaterlis, A. Perez Garcia and B. Genge, "On the Use of Emulab Testbeds for Scientifically Rigorous Experiments," *IEEE Communications Surveys & Tutorials,* vol. 15, no. 2, pp. 929-942, 2013.

[58] E. Eide, R. Ricci, J. Van der Merwe, L. Stoller, K. Webb, J. Duerig, G. Wong, K. Downie, M. Hibler and D. Johnson, "The Emulab Manual: Virtual Machines and Containers," 3 7 2020. [Online]. Available: http://docs.emulab.net/virtual-machines-advanced.html#%28part._docker-containers%29.

[59] Oracle Corporation, "Configurable Resources and Properties for Zones," October 2017. [Online]. Available: https://docs.oracle.com/cd/E53394_01/html/E57855/z.config.ov-3.html#VLZCRgpwbj.

[60] T. Koenig, "sleep(3) — Linux manual page," 22 03 2021. [Online]. Available: https://www.man7.org/linux/man-pages/man3/sleep.3.html.