Precise Virtual Time Advancement for Network Emulation

Abstract

Network emulators enable rapid prototyping and testing of applications. In a typical emulation the execution order and process execution burst lengths are managed by the host platform's operating system, largely independent of the emulator. Timer based mechanisms are typically used, but the imprecision of timer firings introduces imprecision in the advancement of time. This imprecision leads to statistical variation in behavior which is not due to the model.

We describe Kronos, a small set of modifications to the Linux kernel that use precise instruction level tracking of process execution and control over execution order of containers, and so improve the mapping of executed behavior to advancement in time. This, and control of execution and placement of emulated processes in virtual time make the behavior of the emulation independent of the CPU resources of the platform which hosts the emulation. Under Kronos each process has its own virtual clock which is advanced based on a count of the number of x86 assembly instructions executed by its children. We integrate Kronos with the SDN emulator Mininet and experimentally show that Kronos is scalable, in the sense that unlike normal emulation, system behavior is accurately captured even as the size of the emulated system system increases relative to fixed emulation resources. We demonstrate the impact of Kronos' time advancement precision by comparing it against emulations which like Kronos are embedded in virtual time, but unlike Kronos rely on Linux timers to control virtual machines and measure their progress in virtual time.

1 INTRODUCTION

Validation of networked applications is challenging because the system of interest is usually online. If an application is debugged on a live system, errors could be catastrophic to service. Off-line testbeds are used, but the size of the network that can be tested is limited to the resources available in the testbed. A compelling case can be made that the first stage of validation might be done in a virtualized networking environment. This is particularly true when the network is complex, (e.g SDN, P4) [25, 11]. Software updates need to be extensively tested on a staging environment before patches are applied to the production system. A flexible staging environment is necessary to facilitate quick prototyping of complex communication infrastructures and analysis of application interactions over them.

Virtualization techniques like [20, 10, 31, 29, 2] create multiple virtual copies of a physical component and overlay a virtual network over the existing physical resources. Network emulation tools create virtual testbeds. Network emulators like Mininet [3], CORE [9], ns-3 [21] can emulate full network stacks and run containers with applications/programmable switches inside them.

This paper addresses an understudied problem, precision of time advancement in a virtualized network emulation. To better understand the issue, consider the state of practice with network emulations. The execution bursts of virtual machines are typically scheduled by the virtual machine manager, in a round-robin best effort sort of way. "Time" experienced by the virtual machine and emulated network is wallclock time. As the size of the system being emulated increases, the execution of the emulated system cannot "keep up" with real time. A number of research groups have addressed this problem by embedding the emulation in virtual time [17, 33, 36, 24]. Each container has its own virtual clock. It moves ahead in virtual time only while executing, and the executions of the containers are scheduled on the basis of virtual time, so that all containers move forward in virtual time together within a small window of virtual time. The challenge is controlling those execution bursts with precision. If timers are used (for example, TimeKeeper uses kernel hrtimers [16]) the stopping point in instruction stream is non-deterministic, yet the advance in time ascribed to the burst will be the same, regardless. We will see that this variation can have notable impact on the statistical measurements of the system behavior.

One way to achieve instruction level control of execution

bursts is to embed instruction counting in the application code itself. This might be done at the compiler level, but that requires the tools and expertise to create a specialized compiler, access to the source code, and recompilation. It can be done by instrumenting binary executable as was done in [13, 28]. This is very delicate business, and in the case of these two fore-mentioned projects the embedded code is tailored specifically for the computer architectures being simulated. It could be done by stepping the executable an instruction at a time. While this has tremendous flexibility it comes with a tremendously high overhead cost. The technique we describe in this paper is based on instruction counting, but is efficient. Given the objective of executing exactly N instructions our proposed approach takes advantage of the large different in time-scale between processor instruction execution (*ns*) and network latency (μ-sec.) We want to keep processes synchronized at the network level; it takes 120 μ-sec to push a 1500 byte Ethernet frame through a 100 Mpbs interface; a machine executing 1G instructions per second executes 120K instructions in that period of time. So we approach the problem by breaking it up into two pieces. Scheduling an interrupt to occur after M instructions, on servicing the interrupt we find that some $M' \geq M$ instructions have executed. M' is non-deterministic, but empirical studies can establish a probability distribution for $M' - M$, The second phase steps out the remaining $N - M'$ instructions. Called INS-SCHED this mechanism is transparent to the emulated source code and leverages the **ptrace** [8] subsystem and *hardware performance counters* [7] to precisely count and stop a process after the specified number of instructions are executed in a burst. We have modified the open-source TimeKeeper to use INS-SCHED, calling the integration Kronos.

The main contributions of this paper:

- An INS-SCHED implementation which allows a process to be executed for specific number of user-level x86 assembly instructions. We empirically show that the overhead of INS-SCHED is nearly independent of the burst length. This makes sense because with an understanding of the distribution of $M' - M$, given N we can choose M to be close to N, but not so close as to have a chance that $M' > N$. In the case of a 100μ-sec virtual time window, a factor of 3 difference between the speed of the CPU doing the emulation and the speed of the modeled switch, and a 3 GIPs processor performing the emulation, the overhead of controlling the execution is only a factor of 2.5 beyond the native cost of simply executing the instructions.
- A virtual time system called Kronos to control scheduling order of containers, processes within containers and maintain precise virtual time advancement in the emulation. Virtual time advancement uses INS-SCHED.
- An evaluation where programmable P4 switch models [4] are run on a mininet [3] integration with Kronos to demonstrate Kronos's ability to scale emulations of programmable networks with accurate results. The evaluation shows that with Kronos, we can accurately emulate high speed (Gbps) P4 switches with many interconnected hosts on simple VMs.

The rest of the paper is organized as follows. In Section 2, we briefly describe TimeKeeper and benchmark Linux hrtimer accuracy. In Section 3, we describe the INS-SCHED implementation which uses **ptrace** and **perf** subsystems. In Section 4, we detail the architecture and inner workings of Kronos. In Section 5, we present our evaluation which demonstrates Kronos's ability to scale emulations of programmable networks with accuracy, and discuss the support it gives for repeatable experiments. In Section 6, we present related work on this topic and conclude in Section 7.

2 TIMEKEEPER

TimeKeeper [24] is an open source software package comprised of a Kinux kernel module and a small set of changes to the kernel source code. It embeds containers and the processes within them in virtual time. A container or process under TimeKeeper's management is said to be *dilated*. At the start of an emulation, containers are added to TimeKeeper's control by startup scripts. These containers may be hosts, routers or switches which are a part of the emulation. For each dilated container TimeKeeper creates a separate schedule queue to represent and manage processes spawned inside the container. All new processes spawned/forked within a dilated container are automatically detected and added to the container's schedule queue. TimeKeeper bypasses the Linux scheduler and manages the scheduling order and quanta of dilated containers and processes within each container.

TimeKeeper containers are advanced in time-stepped fashion through windows of t units of virtual time. At the start of an emulation window, the virtual clocks of all containers are synchronized to the same value. Processes within a container i are then allowed to run for a cumulative physical duration of $\alpha_i * t$ units where α_i is the Time Dilation Factor (or TDF) of container i. The idea is that α_i models the difference in speed between the CPU executing the emulation, and the CPU being modeled as executing the process. $\alpha_i = 10$ means the modeled computer is ten times faster than the emulation engine, $\alpha_i = 1/10$ means it is 10 times slower. After all containers have run inside the emulation window, the window is forwarded.

During the execution burst, any time related system call (e.g *gettimeofday*, *clock_gettime*) invoked by a dilated process is intercepted and handled based on the container's virtual clock. More complex time based operations like *sleep*, *nanosleep*, *select*, *poll* and *timerfd* are also reinterpreted in virtual time for all dilated processes.

TimeKeeper uses a container specific schedule queue to

Function `Window_Operation(CPU)`**:**

INPUT $\{(container_i, t)\}$
INPUT CPU_MAP
for each container $\in$ CPU_MAP[CPU] :
Get container's execution burst length t
ΔClock(container) = 0
for each process $\in$ container[Queue]:
//t_p = target virtual time advance for process.
1. Compute t_p based on Round Robin schedule among processes inside each container.
2. Resume process execution for $TDF * t_p$ secs.
3. Handle all time related syscalls in virtual time.
4. Freeze process after $TDF * t_p$ secs.
// Advance virtual time of container by t_p units
5. Clock(container) += t_p, ΔClock(container) += t_p.
6. Break loop if ΔClock(container) == t
return

Algorithm 1: TimeKeeper operations during each window

implement round robin scheduling on all child processes. These processes are resumed and paused using kernel level signalling techniques (SIGCONT and SIGSTOP). By using signals, TimeKeeper circumvents the linux scheduler and controls the scheduling order and time-slices allotted to all processes under its control.

Algorithm 1 illustrates how TimeKeeper advances virtual time in each window. It takes as input a set of containers to manage and a mapping specifying the CPU on which each container should run on. The algorithm is run in parallel on all CPUs and repeated in every window.

Suppose a container begins writing a frame at time s, to be received by a different container (which has its own virtual time clock.) If nominally the frame would be received at time $s+L$, and t is the target window size, both TimeKeeper and Kronos compute the smallest integer j such that $s+L \leq j \cdot t$, and the arrival time of the frame is scheduled to be $j \cdot t$. Time $j \cdot t$ falls on the edge of a window; the network emulator implements all communications between VMs to occur while the VMs are suspended and synchronized at a window-edge. Frames delivered at this time are moved to be detectable at the interfaces of their recipients. When the recipient begins running again, the frame presence may cause an interrupt to deal with the arrival. The key point is that TimeKeeper and Kronos artificially increase latency in order to align frame transfers at boundaries of virtual time windows. Correspondingly, to minimize the impact this has on the accuracy of the emulation we typically aim for a small window size. But as we will see, the scale of the imprecision of timer-based virtual time advancement can be of the same order as the desired width of the window. This means that for small values of the physically timed execution burst $\alpha_i t$, variation in the number of instructions executed for a given $\alpha_i t$ can be quite significant. In one window a length $\alpha_i t$ of physical time may result in the execution of b instructions; in another window the burst might execute $2b$ instructions. In both cases the interpreted advance of virtual time, t, is the same.

2.1 Precision of Time Advance

The use of OS hrtimers [16] and signals to control execution bursts lacks precision. A process is signaled to resume, a timer is set to fire after some t microseconds, and on firing the process is signalled to stop. Signals sent from the kernel do not immediately affect a process's execution since they are processed only at select locations in the kernel such as during a return from an interrupt or system call.

To illustrate the issue we set up experiments which measure the difference between a timer epoch and the actual observed time delay. Table 1 contains the mean and standard deviation of such observed hrtimer firing errors for different values of t, measured over hundreds of trials. In each trial, time was noted before starting a hrtimer and immediately after it fires. This mimicks the points at which signals are sent in TimeKeeper to control an execution burst. Due to the observed errors, TimeKeeper is seen to lack precise control over where a process is actually suspended.

Table 1: Mean and Std deviation of hrtimer firing errors.

attempted physical duration (μs)	*mean* (μs)	σ (μs)
100	46.4	6.41
1000	33.15	38.38
10000	86.22	73.71
100000	93.47	69.49
1000000	85.72	65.19

Here we see that trying to control a physical execution burst to be on the order of 100 μ-secs leads to error that on average is almost half that length. Thus, timers are inadequate for fine-grained control of process executions.

3 INSTRUCTION DRIVEN SCHEDULING

In the previous section we noted that TimeKeeper controlled an emulation's advance in virtual time by using unpredictable timer/signal delivery. To eliminate the nondeterminism, we need an approach which tracks the actual execution of a process instead of elapsed time. To accomplish this, we implemented a technique called INS-SCHED which allows a controller process called the *tracer* to assume control over a set of *tracee* processes and (1) track the number of user-level x86 instructions executed by each *tracee* on its designated CPU and (2) precisely pause the *tracee* after it

executes a certain number of instructions, or yields its CPU voluntarily.

Our INS-SCHED implementation interacts with two Linux subsystems: **ptrace** and **perf**. We digress into briefly describing each subsystem's functionality before tying them together with slight modifications to meet our requirements.

3.1 Process tracing with ptrace

ptrace [8] is a process tracing capability built into the Linux kernel to track and control the execution of select processes. It is a system call used by debugging tools such as gdb [5] to assume control over the execution of another user level process (usually referred to as a *tracee*). Debuggers (also referred to as *tracers*) launch the execution of a tracee process with specific ptrace flags. A tracer can then subscribe and listen for specific events (like *clone*, *fork*, signal deliveries etc) over the course of execution of the tracee. Whenever one of these events occur, the tracee is stopped and its tracer is notified. A tracer can also subscribe to another type of event called **single-step** which is triggered after the tracee executes an instruction. A system call is treated as a single instruction and a trigger is issued once after it is finished. Analysis revealed that the single step feature works by setting a Trap flag in the x86 FLAGS register which causes a debug trap exception to be thrown each time an instruction is executed. The trap handler in itself executes thousands of instructions before returning control to the tracer. This is a significant overhead for each executed tracee instruction. Clearly we need an alternate faster approach.

3.2 Hardware performance counters

Performance counters are a set of special purpose registers built into most modern Intel and AMD processors. These registers can be programmed to track counts of hardware events and generate non maskable interrupts (NMI). Several types of performance counters are available in modern processors to count events like cache hits, number of interrupts received, number of mispredicted branches etc. Performance counters can be programmed independently on each processor using the **perf** [7] API exposed by linux.

One of the hardware performance counters tracks the **number of instructions retired or completed**. This counter can also distinguish between instructions executed in user mode and kernel mode using the value of current privilege level (CPL) register. It can be programmed with an initial value and type of instructions (user/kernel) to count. As each instruction is executed, the processor decrements the counter and an NMI is generated on reaching zero. It is then reset to its original value. It can be used by a tracer process to *poll* and wait until a certain number of user-level instructions are executed by its tracee.

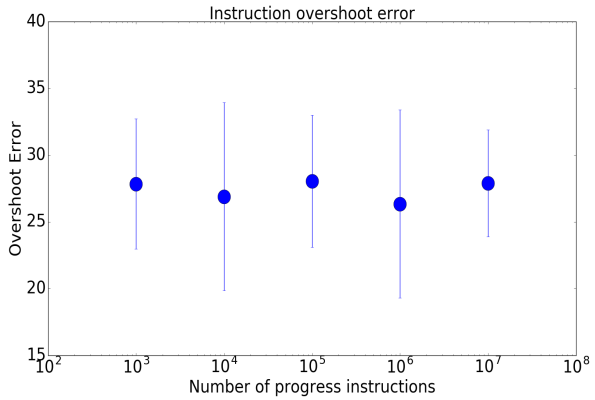


Figure 1: Observed counter over-shoot errors

However, there are a few drawbacks in using **perf** to implement INS-SCHED. We outline them here.

Counter overshoot error: The *poll* mechanism used by the tracer works internally by waiting for a file descriptor to become active. After the counter reaches zero and invokes an NMI interrupt, the interrupt handler uses the Linux *irq* work queue to schedule another *irq_work* interrupt to be triggered by the CPU. The triggered *irq_work* interrupt activates the file descriptor polled on by the tracer process. To ensure that the tracee doesn't execute in another core while *irq_work* and NMI interrupts are being serviced, we confined its affinity to a specific CPU and programmed counters to fire only on that CPU. We instrumented the Linux kernel code to verify the counter value at the time of activation of the file descriptor and observed some overshoot error.

Function `run_process`$(tracee, N, \delta, CPU_i)$**:**
  if $N \leq \delta$ **then**
    initiate_multistep($tracee, \delta$)
  else
    $tracee.n\_ints = 0$
    $tracee.ptrace\_msteps = 0$
    Program CPU_i's perf counter to fire at $N - \delta$
  end if
  Wait for wakeup from tracee
return

Algorithm 2: The INS-SCHED API invoked by a tracer to run a specific tracee for $N - \delta$ instructions on CPU_i

Figure 1 plots the counter overshoot error as the target instruction count is varied. It shows that there is a fairly stable non zero overshoot error for different target counts. Further analysis revealed that there is some time difference between the NMI and *irq_work* interrupt firings and the tracee continues to execute during this time. This additional

```
Function initiate_multistep(tracee, n_steps):
    tracee.ptrace_msteps = n_steps
    Set TRAP flag and continue tracee
return

Function debug_trap():
    if tracee.ptrace_msteps > 0 then
        tracee.ptrace_msteps − −
    else
        Clear TRAP flag and send wakeup to tracer
    end if
return

Function on_interrupt_exception_entry():
    if interrupt/exception arrives in user mode then
        tracee.n_ints + +
    end if
return

Function on_perf_interrupt():
    v ← read counter value
    single_steps ← N − v + tracee.n_ints
    initiate_multistep(tracee, max(1, single_steps))
return

Function schedule():
    if !preempt && tracee.enter_syscall then
        Send wakeup to tracer
    end if
return
```

Algorithm 3: Relevant portions of the INS-SCHED logic embedded into the Linux Kernel.

uncertainty is added to the system because **perf** does not provide any support to precisely stop a tracee from resuming execution after its instruction count has expired.

Counter value error: Another challenge with using **perf** involves erroneous count values reported by the hardware counter. All interrupts and exceptions which occur during a tracee's execution are included in the instruction count values. Thus, if hundred interrupts occur during a tracee execution burst, each interrupt is counted as an instruction executed by the tracee. Since the tracee does not have control over how many interrupts/exceptions occur during its execution, the reported counter values are erroneously offset by interrupt and exception counts. However, we are able to detect these errors and remove the impact they have on the target number of instructions executed.

3.3 Kernel modifications

In the previous two subsections, we outlined challenges in using **ptrace** or **perf** to implement INS-SCHED. In addition to being precise, an INS-SCHED implementation should also detect voluntary CPU yields from any tracee. We only briefly stated this requirement before but its importance is significant because a tracer should never be blocked *forever* waiting for its tracee to finish its execution burst. For instance, a tracee may try to acquire a semaphore and sleep if the semaphore is held by another tracee. The tracer should be notified of this event so that it is not blocked forever.

All of our modifications were made to the Linux kernel version 4.4.5. We added two variables to the *task_struct* process descriptor: *ptrace_msteps* and *n_ints*. The variable *n_ints* is a counter, which is incremented on each interrupt and exception entry in *arch/x86/entry/entry_64.S* inside the linux kernel, to correct for the inflated instruction counts when we engage in single-stepping mode. The variable *ptrace_msteps* holds the number of instructions to be single-stepped.

In our design, the tracer confines a tracee to a specific CPU and programs that CPU's hardware instruction counter to fire at a value which is close to but less than the target value and initiate *multi-steps* for the remainder of the execution burst. This initiation is described by function `run_process` in Algorithm 2. If the total number of steps desired N is less than the maximum number of single steps (δ) we shift immediately into single stepping mode (calling function `initiate_multistep`.) Otherwise, the normal course of logic, the selected CPU's performance counter is set to cause an interrupt after $N - \delta$ instructions. The count of the number of interrupts taken (*tracee.n_ints*) and the number of instructions to single-step at this point (*tracee.ptrace_msteps*) are both set to zero.

During the course of the $N - \delta$ instructions it may happen that interrupts and exceptions arrive. The performance counter adds one instruction each time this happens (to account for taking the interrupt/exception, even though the instructions executed in the interrupts and exceptions are not themselves counted.) Accordingly, as shown in function `on_interrupt_exception_entry`, the counter *tracee.n_ints* is incremented with each interrupt and exception, which will allow us to correct for the inflated instruction count when we engage in single-stepping mode.

Function `on_perf_interrupt` fires after the scheduled $N - \delta$ instructions are executed, although as we have noted there will be some over-shoot. We read the actual number of instructions executed (a count which is too large by as many interrupts as were taken during the execution) and compute the remaining number of instructions needed to execute to achieve the goal of precisely N. Function `initiate_multistep` is called to start the instruction-by-instruction execution, accomplished by setting the tracee's

ptrace_msteps variable to the number of instructions to single-step. Thereafter the execution of every instruction causes a trap, handled by function `debug_trap`, where, once the counter of instrutions to step through reaches zero, a wake-up call is passed to tracer to indicate the completion of the N instruction executions.

Anywhere in this sequencing it may happen that the CPU is voluntarily yielded. If a tracee yields its CPU voluntarily during an execution burst, its tracer should be immediately notified. A tracee may relinquish its CPU for many reasons (e.g waiting for an I/O to complete, or waiting for a semaphore to become available) and it may remain blocked for a long time. A tracer however should not be blocked forever waiting for its tracee to complete its execution burst.

In the Linux kernel, processes voluntarily yield their CPU by calling the `schedule()` function which invokes the Linux scheduler and swaps out the running process. But under normal operating conditions, the scheduler is also invoked on return to *user* mode from a syscall/interrupt if a resched flag (TIF_NEED_RESCHED) is set (upon quanta expiry) by a tick timer interrupt. It may also be invoked if the kernel decides to pre-empt a process.

We need to distinguish both cases from a voluntary CPU relinquishment. A pre-emptive schedule invocation can be easily identified because it sets a kernel level *preempt* flag in the scheduler implementation. To identify tick interrupt driven scheduler invocations, we set a per-process *enter_syscall* flag upon syscall entry (in *arch/x86/entry/common.c*) and clear it on return prior to the check on resched flag. Since all checks on the resched flag are made only prior to a return to *user* mode, our modifications ensure that *enter_syscall* flag will never be set during a resched flag/tick interrupt driven scheduler invocation.

Thus, inside the function (Algorithm 3: schedule) we send notifications to the tracer only if the tracer is inside a syscall and was not pre-empted. The *enter_syscall* flag is cleared atomically when the tracee becomes runnable again. A tracer would be able to test for the flag at a later stage and determine if its tracee is still blocked.

The net effect of a voluntary CPU yield is to credit the process for executing N instructions, even though fewer were actually executed. In principle we might improve accuracy by doing a better job of determining precisely how many instructions were executed at the point of the yield, but as this impacts scheduling as well, leave this to future work.

3.3.1 Benchmarking

There are four steps to an INS-SCHED controlled burst. First, the **perf** mechanism is initialized, being told how many instructions M to trace. Secondly, the instruction stream begins to execute. After *approximately M* instructions the third step engages wherein **perf** generates an interrupt, the processing of which measures precisely how many instructions executed between the **perf** setup and the interrupt. Noting that some M' instructions were executed, and that a target number N are desired, the final step executes the remaining $N - M'$ instructions one at a time, in single-step mode.

We are interested in three execution costs. First, the per burst cost of setting up **perf**. Second, the native per-instruction cost of executing instructions. This cost helps us interpret the scale of the INS-SCHED overheads. The final cost of interest is the per-step cost of single stepping the execution to close exactly in on executing precisely N instructions. Table 2 lists the average results of several experiments, each run 2000 times on a Intel i7 sixth generation processor with eight cores, running at 2.7GHz. Each experiment sets up **perf** to execute some N instructions, runs an instruction stream until the **perf** interrupt, then steps through some K individual steps. The columns of the table describe N, ranging from 1000 to 10,000,000 by powers of 10. The rows of the table describe K, ranging from 0 to 500. An experiment is not exactly like the application of INS-SCHED, because in application the number of single steps is variable. However, these experiments give us data we can analyze to estimate costs. The costs are linear in N and K and we can use linear regression to estimate them. The per-burst setup cost of **perf** is close to 180 μ-secs, the time per instruction is 0.32 *ns*, and the cost per single step is close to 400 *ns*. The variation in number of instructions actually controlled by **perf** is so small that choosing M so that $N - M = 50$ always led to $M' < N$, which means that planning to single step around 50 instructions) is safe. A thumbnail sketch then of comparative overheads shows that if the synchronization window is 1,000,000 instructions (which translates to a physical window size of 320 μ-sec on the test machine, corresponding to a virtual window size of 106.67 μ-sec when the TDF is 3), and up to 50 of the instructions are single-stepped, then the time to execute the controlled burst is 320 μ-secs, which 320 μ-secs is native instruction execution. This means the overhead induces a slow-down over native execution of just 1.6. Of course, the specific slow-downs will depend on a number of factors, but this sketch shows that in realistic scenarios the overhead is not large.

Table 2: Average execution time (μ-secs) for controlled instruction bursts

K/N	10^3	10^4	10^5	10^6	10^7
0	181.92	182.81	213.22	503.17	3408.97
50	200.18	203.39	234.22	522.01	3415.30
100	210.10	216.65	243.61	537.11	3442.82
200	254.46	255.31	287.43	574.13	3495.53
500	378.57	380.12	409.35	697.96	3618.22

4 KRONOS

Kronos is a Linux virtual time system which uses INS-SCHED discussed in Section 3 to control execution bursts. Its design is inspired from TimeKeeper.

Kronos is a Linux kernel module which can control the execution order of a set of registered container processes. Emulation setup scripts start containers and launch generic tracer processes inside each container. Each tracer process is specified with a set of commands to launch inside the container. These commands could include host, switch or router processes. The tracer process in each container registers itself with Kronos, and uses ptrace to launch and freeze tracees . Kronos advances the emulation in windowed time-steps similar to TimeKeeper. It exposes a set of user facing APIs which can be used by the emulation setup script and individual tracer processes to report progress or make configuration changes during the emulation. The Kronos API is briefly summarized below:

- *progress(n)*: This function can be called by an emulation setup script to run the emulation for n consecutive windows before returning control to it.
- *update_container(i,t,relative_cpu_speed)*: This function can be called by an emulation setup script before the start of a window. It instructs container i to advance by t units of virtual time during every subsequent window. But in Kronos, since execution bursts of containers are specified in units of instructions, the third argument *relative_cpu_speed* converts window size t into execution burst lengths (in instructions) for all subsequent windows. A *relative_cpu_speed* of k implies that k instructions can be executed in $1ns$ of virtual time.
- *get_window_config(i)*: It is invoked by the tracer in container i at the start of every window and it returns the total number of instructions to be executed by the tracer during that window.
- *update_clock(i,δ)*: It is invoked by the tracer in container i to inform Kronos that it has executed δ instructions in the current window. Kronos then updates the container's clock based on its assigned cpu_speed.
- *signal_finish()*: It is invoked by a tracer to signal the end of its window execution. The tracer is blocked until all other tracers finish executing their windows.

Kronos intercepts time related computations during each tracee's execution burst and handles it based on a container's virtual clock. It intercepts system calls which query time like *gettimeofday*, *time*, *clock_gettime* and other system calls which reference time for some computation like *sleep*, *select*, *poll*, *timerfd* and *itimer*. For the sake of brevity we do not describe all the associated Kronos specific kernel modifications here. Many of them were adopted from TimeKeeper and optimized to reduce overhead and work better with INS-SCHED. Algorithm 4 depicts a typical emulation setup script and how a tracer operates during each window.

```
Function Tracer_Operation(i):
    Register with Kronos
    while Emulation is not stopped do
        n = get_window_config(i)
        ρ ← Scheduling policy, δ ← 50
        for each tracee in Tracer's control do :
            //Find tracee's quanta N based on ρ
            Compute N = ρ(tracee, n)
            if N > 0 then
                run_process(tracee, N, δ) //INS-SCHED
                update_clock(i,N)
            end if
        end for
        signal_finish()
    end while
return

Function Emulation_Setup(N_Rounds, N):
    Start all N tracers
    Wait for all tracers to register with Kronos
    for i in 1 to N_Rounds do :
        progress(1)
        for j in 1 to N do :
            if necessary, update_container(j, t, cpu_speed)
        end for
    end for
return
```

Algorithm 4: Interactions between Kronos, tracers and emulation setup scripts

5 EVALUATION

In our evaluation, we demonstrate Kronos's ability to emulate programmable P4 networks while providing accurate results, even on large networks. We used mininet [3], augmenting it to run Bmv2 P4 switches [4] instead of the OVS switch [6] that normally accompanies mininet distributions. The modified emulation startup scripts initiate a simple shortest path forwarding P4 program on each Bmv2 switch and programmed forwarding rules on each switch through the P4 thrift service.

All of our experiments were performed on a Xeon server running Ubuntu 16.04 with 56 cores clocked at 2.0 GHz and 64 GB of RAM. Worker VMs were spawned inside the server and each VM was allotted 8 cores and 12 GB of RAM. Each ran a modified Linux kernel v4.4.5 to support Kro-

nos and TimeKeeper. Unless otherwise stated, all experiments with Kronos and TimeKeeper were performed with a *relative_cpu_speed* (or time-dilation factor) of 3. All links were assigned a delay of $1ms$ and a fixed window step size $t = 100\mu$-sec was used in all Kronos and TimeKeeper controlled experiments.

5.1 Evaluating Scale

In this experiment, we demonstrate Kronos's ability to support emulations with many hosts and switches without compromising on the fidelity of experiment results. We use a topology containing an even number of switches ($2n$) connected in a ring with one host attached to each switch. The hosts and switches are numbered increasingly (from 1 to $2n$) in clockwise order. Iperf clients (servers) are started in all odd (even) numbered hosts respectively. Each client sends udp traffic at 300Mbps to its clockwise neighbouring server.

This experiment setup is designed to create n non-interfering flows (using the network bandwidth measurement tool *iperf*) between neighbouring client-server hosts. Because of the non-interference, any significant degradation from the 300Mbps transfer is due to the emulation not being able to "keep up" in real time. To increase the size of the topology and the number of flows we increased n from 5 to 25. The observed average data transfer rates across all n iperf flows are compared in Figure 2a. Figure 2b compares the cummulative distribution function of the observed data transfer rates across all topology sizes. These graphs illustrate that best-effort emulation is unable to maintain a data transfer rate of 300 Mbps as the size of the topology increases. But Kronos is able to deliver 300 Mbps data transfer rates (in virtual time) for all iperf flows regardless of the topology size.

5.2 Impact of Time Advancement Accuracy

In the next experiment, we illustrate the impact that time advancement inaccuracy has on TimeKeeper's behavior, and how that is corrected using INS-SCHED and Kronos. We build a 2 level fat tree topology with $fanout = 5$ to link all switches. Each leaf switch is linked with two hosts. This topology is illustrated in Figure 3; it has 31 switches and 50 hosts. The hosts are numbered left to right from 1 to 50. Simple UDP clients (servers) are started on hosts $h1 - h25$ ($h26 - h50$) respectively. Repetitive UDP request-response connections are initiated between every i^{th} UDP client and i^{th} UDP server. All paths go through five switches (including the root) and each switch manages multiple flows. Statistically the round-trip delays of each flow have the same probability distribution. Figure 4 plots the empirical cumulative distribution function of UDP round-trip delays under Kronos, TimeKeeper, and Best Effort time management systems. The nearly vertical form of the Kronos curve says that Kronos observes very little variance per UDP flow. The spread of the TimeKeeper curve from values as low as 12 ms to as high as 17 ms for the same model reflects that the variation on measured virtual time is due to the inaccuracies of timer-based measurement, which is not a problem with INS-SCHED and Kronos. The larger values and spread of Best Effort shows the impact that lack of explicit control over virtual time has on observed behavior.

5.3 Varying relative CPU speed

The relative CPU speed of a container is the conversion ratio applied to translate the number of instructions executed by the container into elapsed virtual time. In other words, it is an emulation parameter which specifies the speed of the cpu perceived by a process running inside a container. For instance, a real hardware P4 switch might execute k instructions per nano second. This information can be incorporated into the emulation by assigning a cpu speed of k to all containers running software P4 switches.

Note that the relative CPU speed of each container should be decided prior to the start of an emulation in accordance with the physical attributes of the production system. Nevertheless, in this experiment we demonstrate Kronos's ability to reproduce expected behaviour of high speed switches.

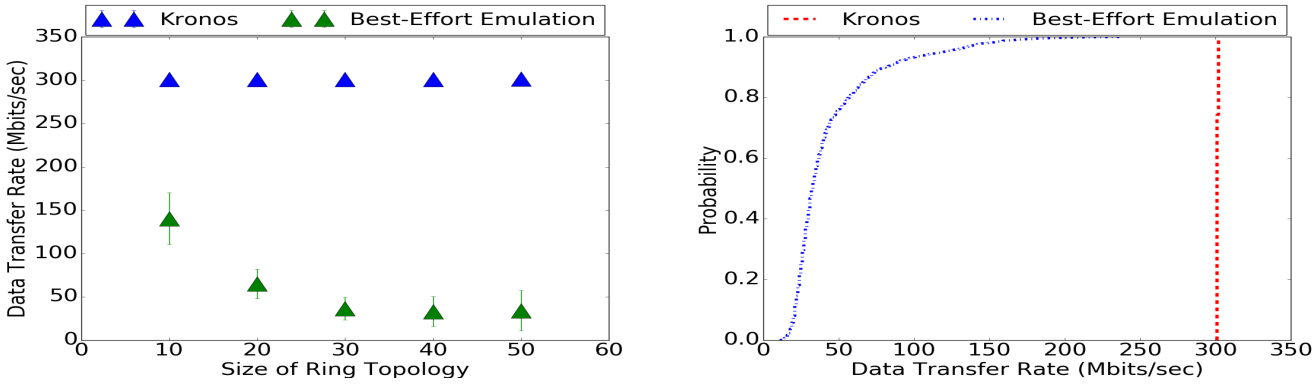
If the relative CPU speed of a Bmv2 switch is increased, it can execute more instructions per unit of virtual time advance and thus process more packets. To verify this claim, we use a star topology consisting of 1 P4 switch connected to 64 hosts. 32 Iperf UDP flows are initiated between host-pairs with data transfer rates set to 500 Mbps for each flow. In Figure 5a, we show the cdf of throughput observed among all flows as the relative cpu speed is increased. It depicts that the switch is able to equally service each flow and the observed data transfer rate for each flow increases as the relative cpu speed of the switch is increased. By summing up the data transfer rate provided to each flow, we plot the cumulative throughput provided by the switch in Figure 5.

We have thus shown Kronos's ability to faithfully emulate high speed (Gbps) P4 switches on a simple VM.

5.4 Repeatability

The precision INS-SCHED brings to virtual time advancement has in certain circumstances the additional benefit of *repeatability*, when a exact repetition of the same experiment yields exactly the same set of events at exactly the same times. The imprecision of timer-based virtual time advancement negates any possibility of repeatability, so defined. However use of INS-SCHED supports repeatability.

To demonstrate the point we wrote a program to recursively compute Fibonacci numbers. Over many experiments we measured the length of virtual time to compute the $1,000,000^{th}$ number. Under INS-SCHED and Kronos,



(a) (b)

Figure 2: Observed data transfer rates for non-interfering flows in ring topology: (a) Comparison of average data transfer rates with increase in topology size. (b) CDF of observed data transfer rates across all flows and all topology sizes.

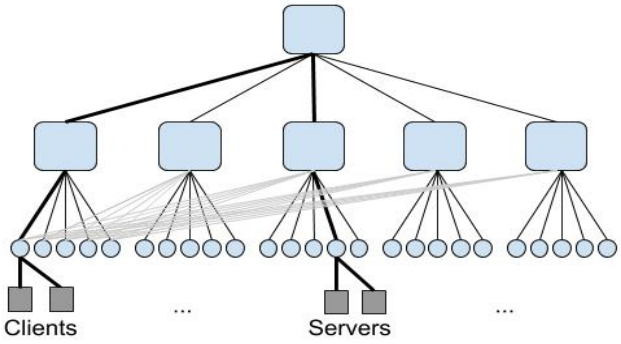


Figure 3: Two-level Fat-tree with fanout of 5

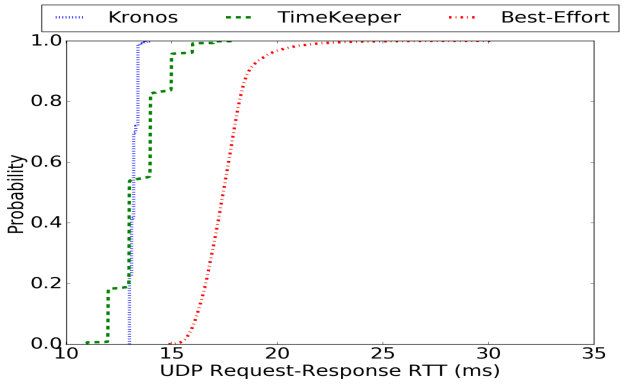
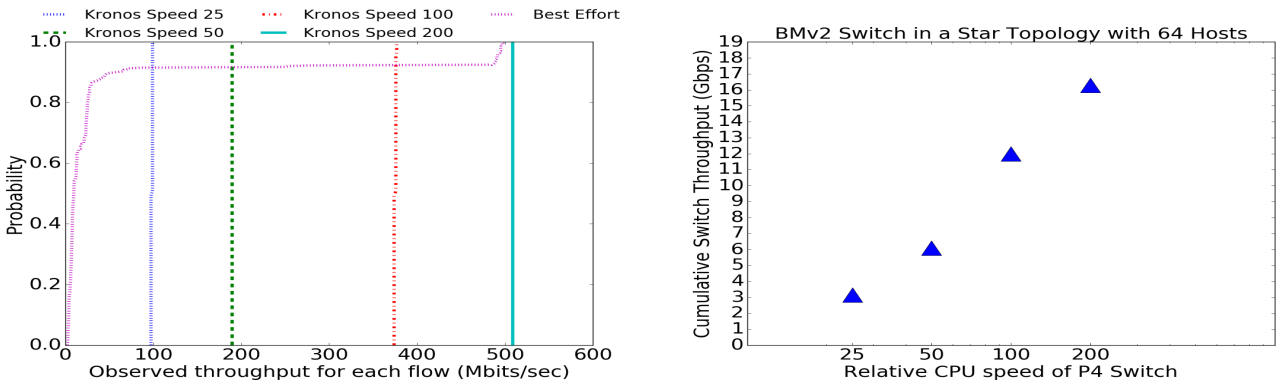


Figure 4: Evaluating accuracy: comparing UDP round trip times across 3 experiment runs in a fat tree topology with fanout = 5.

for every experiment the delay was exactly the same, the number of instructions executed. Performing the same experiment with TimeKeeper gives varying results. The coefficient of variation (ratio of standard deviation to mean) of elapsed virtual time was observed to be 0.17 ms. To demonstrate the potential impact of timing variation on the actual execution path followed, we modified the Fibonacci code to take a branch and raise an exception if the length of time elapsed to compute a particular Fibonacci number exceeded some selected threshold. Under INS-SCHED and Kronos the branch was never taken; under TimeKeeper it was taken roughly 20% of the time.

Another way to quantify repeatability, at least in a statistical sense, is to compare the empirically collected distribution functions of time-measured metrics, such as round-trip delay. Imagine running the same experiment several times under INS-SCHED and Kronos, and running that same experiment several times using TimeKeeper, and using best effort scheduling. If random number seeds are synchronized each run, then if we had perfect repeatability we would observe exactly the same events and obtain for each run exactly the same empirical distribution function. The degree to which we do not see exactly the same distribution is a measure of the non-determinism introduced by lack of time advance precision. We use a well-known statistic for comparing distributions, the Kullback-Liebler divergence metric [1]. The KL metric derived from two distributions is an information-theoretic measure of the difference in information between the two distributions, the number of bits per sample needed to describe the difference. Small values imply that the distributions are very close to each other. Table 3 reports the KL divergence metric derived from comparing pairs from three runs which measure round-trip delays of communications in the fat-tree example. Metrics from Kronos and INS-SCHED (Kr), TimeKeeper (TK), and Best Effort (BE) are reported. These statistics show a very clear trend, that the KL divergence among the Kronos runs is exceedingly small, TimeKeeper's is at least an order of magnitude larger, and Best Effort divergence is the largest of all. However, the Kronos runs are not *perfectly* the same, a point we next take up.

An emulated program may achieve functional repeatability even if the time advancement is not precise. The danger comes when the program's output depends on time, for example, computing the round-trip delay, as we have seen, or



(a) (b)

Figure 5: Data transfer rates as relative cpu speed is varied in a star topology: (a) Comparison of CDF of iperf data transfer rates across all 32 flows at different relative cpu speeds. (b) Cumulative data transfer rates provided by the switch as its relative cpu speed is increased.

Table 3: Pairwise KL_{ij} divergence between round trip time distributions of each of the 3 experiment runs of the fat tree example

	KL_{12}	KL_{13}	KL_{21}	KL_{23}	KL_{31}	KL_{32}
BE	0.26	0.119	0.112	0.092	0.103	0.094
TK	0.033	0.053	0.032	0.092	0.044	0.075
Kr	6.7e-4	2.8e-3	2e-3	2.5e-4	4.2e-3	1.1e-3

if a program branch depends on time, e.g., polling a socket. In such contexts the precision of INS-SCHED is an aid to proper understanding of system behavior.

However, even using INS-SCHED it is possible to not have repeatability in this exact sense. This sometimes occurs where a process voluntarily yields its CPU at unpredictable moments. For instance, during an execution burst, a process writing to a file would invoke the *write* system call. The system call handler might yield the CPU temporarily because the hard disk might be busy. Since our INS-SCHED implementation is designed to prematurely return following such voluntary CPU yields, the process is taken off the run queue for the remainder of the emulation window. Such temporary CPU relinquishment may not be triggered during the next emulation run.

Thus, our inability to differentiate CPU yields due to external events from those tied to a process's execution can introduce some non-deterministic behavior. One approach to mitigate this problem would be to white-listing the set of system calls which are not normally expected to yield the CPU and ignore all notifications from calls to *schedule()* invoked within the context of any such white listed system call. Another approach to solve the problem could involve instrumenting the source code of the emulated application to instruct Kronos prior to a system call entry on whether it expects to yield its CPU inside the syscall handler.

Both approaches have their own drawbacks. White listing system calls does not completely eliminate the problem while instrumenting the source code of the emulated application can be time consuming. We leave the addressing of this problem for future work.

6 RELATED WORK

Related work in this space is mainly divided into 3 categories: virtual time driven emulation, virtual testbeds and program execution replay.

Virtual time driven emulation: The concept of embedding virtual time in emulation is not new. In [19, 18], Gupta et. al put forward the notion of time dilation as a potential solution to improve scalability of hybrid emulation-simulation systems. They defined Time Dilation Factor (TDF) as the ratio of rate of progress of real time to virtual time. A protocol evaluation testbed called SVEET! [15] was built using the proposed time dilation technique and used to evaluate TCP implementations. This work emphasizes the importance of virtual time systems in performance analysis of emerging technologies. The testbed accurately predicted TCP performance on slower hardware and demonstrated the cost benefits of time dilation.

Zheng et. al in [36] adopt a new approach to advancing virtual time which unlike Gupta's solution, is less tied to the advancement of real time. The proposed virtual time system includes a virtual time controller to decide how far each container should advance in virtual time. The virtual time controller would maintain virtual synchrony among all emulated containers by blocking containers which have advanced too far in virtual time to allow others to catch up.

Zheng's work was extended by Jin et. al in [22] specifically to simulate software defined networks (using OpenVZ) and in [35] using Linux containers. TimeKeeper [24] builds on these works by bringing the notion of TDF to the forefront and allows more finer control over process execution

times with small modifications to the Linux Kernel.

Virtual testbeds: An emulated approach to system study uses virtualization techniques like [29, 31, 10, 2], and is a common means of understanding models. Three very widely used emulation testbeds are Emulab [34], DETER [32], and PlanetLab [30]. In Emulab, the experimenter can create the desired network and gain access to the nodes within the testbed for a period of time. The experimenter is given admin access to all virtual nodes and may install any OS on them. PlanetLab is a collection of connected machines around the world. Deter is an emulation platform built over Emulab with a is focus on cyber-security. PlanetLab gives the experimenter a LXC container on each node in the experiment. Therefore, the user cannot run customize the OS, and will compete for system resources with other users on the same node.

Replaying program execution: Previous works in this space [14, 27, 12, 26, 23] focus on logging events during a program's execution and replaying them. These techniques can incur significant space and time overheads. In Revirt [14], performance counters are used to record instructions interrupted by asynchronous events like interrupts. PinPlay [27], extends replaying capability to parallel programs and threads by recording register and memory contents at checkpoints. In [23], system call return values are stored and returned upon execution. ExecRecorder [12] replays the execution of an entire VM to facilitate intrusion analysis. It uses checkpoints containing complete system state (virtual memory, CPU registers, hard disk and memory of virtual external devices) to replay VM execution. BugNet [26] continuously tracks and stores register state of an application and enables replaying execution across interrupts.

7 CONCLUSION

Network emulations are now commonly embedded in virtual time. This simplifies their integration with discrete-event simulators, and allows realistic behaviors to be observed even when the size of the model being emulated is large compared to the resources of the emulator. However, to embed process execution in virtual time it is necessary to ascribe lengths of virtual time to execution bursts, so that one may say "this process now executed for *s* virtual seconds". In this paper we consider the issue of precisely ascribing that virtual duration to an execution burst. We show that timer-based mechanisms of the past are quite imprecise at time-scales of interest to us (minimum network latency), and show that this imprecision can have impact on the observed system behavior, e.g., the probability distribution of round-trip delays. We propose a mechanism for precisely ascribing virtual time to execution bursts, by choosing the number of instructions to have executed in a burst, and to arrange control so that exactly that number of instructions are executed. The technique we propose, INS-SCHED is computationally feasible when the burst length can be measured in 10s or 100s of thousands of instructions. INS-SCHED works by setting up an interrupt to occur after an instruction counter reaches a particular threshold, set to be just shy of the actual number of instructions we desire to execute. A finishing phase single steps the remaining number of instructions.

Using network models involving complex programmable P4 switches we demonstrate the ability of INS-SCHED and the Kronos system which uses it to accurately model flows in models of moderate size (10s of switches). We also comment on the potential for INS-SCHED to support perfectly repeatable emulations, identifying circumstances where it presently does not, but also identifying potential future approaches that might bring perfect repeatability.

References

[1] On information and sufficiency. *The Annals of Mathematical Statistics 22*, 1 (March 1951), 79–86.

[2] Openvz: a container-based virtualization for linux. http://openvz.org/Main*Page*, 2014.

[3] mininet: An instant virtual network on your laptop. http://mininet.org/, 2015.

[4] Bmv2 - behavioural model software p4 switches. https://www.bmv2.org, 2018.

[5] Gdb: The gnu project debugger. https://www.gnu.org/software/gdb/, 2018.

[6] Open vswitch. https://www.openvswitch.org, 2018.

[7] Perf: The linux performance counter subsystem. https://perf.wiki.kernel.org/index.php/Main_Page, 2018.

[8] Ptrace: The linux process tracing subsystem. https://linux.die.net/man/2/ptrace/, 2018.

[9] AHRENHOLZ, J., DANILOV, C., HENDERSON, T. R., AND KIM, J. H. Core: A real-time network emulator. In *Military Communications Conference, 2008. MILCOM 2008. IEEE* (2008), IEEE, pp. 1–7.

[10] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating System Principles* (2003).

[11] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., ET AL.

P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review 44*, 3 (2014), 87–95.

[12] DE OLIVEIRA, D. A., CRANDALL, J. R., WASSERMANN, G., WU, S. F., SU, Z., AND CHONG, F. T. Execrecorder: Vm-based full-system replay for attack analysis and system recovery. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability* (2006), ACM, pp. 66–71.

[13] DICKENS, P., NICOL, D., AND HEIDELBERGER, P. Parallelized direct execution simulation of message passing programs. *IEEE Transactions on Parallel and Distributed Systems 7*, 10 (October 1996), 1090–1105.

[14] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 211–224.

[15] ERAZO, M. A., LI, Y., AND LIU, J. Sveet! a scalable virtualized evaluation environment for tcp. In *Testbeds and Research Infrastructures for the Development of Networks & Communities and Workshops, 2009. TridentCom 2009. 5th International Conference on* (2009), IEEE, pp. 1–10.

[16] GLEIXNER, T., AND NIEHAUS, D. Hrtimers and beyond: Transforming the linux time subsystems. In *Proceedings of the Linux symposium* (2006), vol. 1, Citeseer, pp. 333–346.

[17] GRAU, A., MAIER, S., HERRMANN, K., AND ROTHERMEL, K. Time jails: A hybrid approach to scalable network emulation. In *Principles of Advanced and Distributed Simulation, 2008. PADS'08. 22nd Workshop on* (2008), IEEE, pp. 7–14.

[18] GUPTA, D., VISHWANATH, K. V., MCNETT, M., VAHDAT, A., YOCUM, K., SNOEREN, A., AND VOELKER, G. M. Diecast: Testing distributed systems with an accurate scale model. *ACM Transactions on Computer Systems (TOCS) 29*, 2 (2011), 4.

[19] GUPTA, D., YOCUM, K., MCNETT, M., SNOEREN, A. C., VAHDAT, A., AND VOELKER, G. M. To infinity and beyond: time warped network emulation. In *Proceedings of the twentieth ACM symposium on Operating systems principles* (2005), ACM, pp. 1–2.

[20] HELSLEY, M. Lxc: Linux container tools. *IBM developerWorks Technical Library* (2009).

[21] HENDERSON, T. R., ROY, S., FLOYD, S., AND RILEY, G. F. ns-3 project goals. In *Proceeding from the 2006 workshop on ns-2: the IP network simulator* (2006), ACM, p. 13.

[22] JIN, D., AND NICOL, D. M. Parallel simulation and virtual-machine-based emulation of software-defined networks. *ACM Trans. Model. Comput. Simul. 26*, 1 (Dec. 2015), 8:1–8:27.

[23] LAADAN, O., VIENNOT, N., AND NIEH, J. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *ACM SIGMETRICS performance evaluation review* (2010), vol. 38, ACM, pp. 155–166.

[24] LAMPS, J., NICOL, D. M., AND CAESAR, M. Timekeeper: a lightweight virtual time system for linux. In *Proceedings of the 2nd ACM SIGSIM/PADS conference on Principles of advanced discrete simulation* (2014), ACM, pp. 179–186.

[25] MCKEOWN, N. Software-defined networking. *INFOCOM keynote talk 17*, 2 (2009), 30–32.

[26] NARAYANASAMY, S., POKAM, G., AND CALDER, B. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ACM SIGARCH Computer Architecture News* (2005), vol. 33, IEEE Computer Society, pp. 284–295.

[27] PATIL, H., PEREIRA, C., STALLCUP, M., LUECK, G., AND COWNIE, J. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization* (2010), ACM, pp. 2–11.

[28] REINHARDT, S. K., HILL, M. D., LARUS, J. R., LEBECK, A. R., LEWIS, J. C., AND WOOD, D. A. The wisconsin wind tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 1993), SIGMETRICS '93, ACM, pp. 48–60.

[29] ROSENBLUM, M. Vmwares virtual platform. In *Proceedings of hot chips* (1999), vol. 1999, pp. 185–196.

[30] VAHDAT, A., YOCUM, K., WALSH, K., MAHADEVAN, P., KOSTIĆ, D., CHASE, J., AND BECKER, D. Scalability and accuracy in a large-scale network emulator. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 271–284.

[31] WATSON, J. Virtualbox: bits and bytes masquerading as machines. *Linux Journal 2008*, 166 (2008), 1.

[32] WEI, S., KO, C., MIRKOVIC, J., AND HUSSAIN, A. Tools for worm experimentation on the deter testbed. In *2009 5th International Conference on Testbeds and Research Infrastructures for the Development of Networks Communities and Workshops* (April 2009), pp. 1–10.

[33] WEINGÄRTNER, E., SCHMIDT, F., VOM LEHN, H., HEER, T., AND WEHRLE, K. Slicetime: A platform for scalable and accurate network emulation. In *NSDI* (2011).

[34] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 255–270.

[35] YAN, J., AND JIN, D. A lightweight container-based virtual time system for software-defined network emulation. *Journal of Simulation 11*, 3 (2017), 253–266.

[36] ZHENG, Y., NICOL, D. M., JIN, D., AND TANAKA, N. A virtual time system for virtualization-based network emulations and simulations. *Journal of Simulation 6*, 3 (2012), 205–213.