Submitted for publication. Author Copy - do not redistribute.

A Parallel Network Simulation and Virtual-Time-Based Network Emulation Testbed

Dong Jin Yuhao Zheng David M. Nicol

dongjin2@illinois.edu zheng7@illinois.edu dmnicol@illinois.edu

University of Illinois at Urbana-Champaign

Urbana, Illinois, USA

Abstract—To analyze large-scale systems with high fidelity, it is necessary for a network testbed to offer both realistic emulation (to represent software execution) and effective simulation (to model background computation and communication). We present a network testbed that integrates a light-weighted emulation system OpenVZ (modified earlier to operate in virtual time) with a parallel discrete-event network simulator S3F. Our algorithmic contributions lay in the design and management of virtual time as it transitions from emulation, to simulation, and back. We also address the unavoidable uncertainties involved in emulation behavior by finding analytical bounds for the error, and produced empirical data showing that the error is as small as the minimum system execution unit. In addition, we observe excellent system scalability with large-scale network experiments, and study the performance impact of the size of the synchronization window, which serves as a strong motivation for developing efficient emulation lookahead in our testbed.

Keywords—parallel discrete event simulation; network emulation, virtual time

1. Introduction

The advancement of large-scale computer and communication networks, such as Internet and power grid control networks, heavily depends on the successful transformation from in-house research efforts to real productions. To enhance this transformation, research has created various network testbeds that use emulation, or simulation, for conducting medium to large scale experiments. The emulation testbeds coordinate real physical devices and provide a configurable environment to conduct live experiments, but for networking are constrained by budget and what can be equipped in a lab. This limits scalability and flexibility. On the other hand, network simulation provides better scalability and much more flexibility, but degrades fidelity owing to the

sort of model abstraction and simplification necessary to achieve scale. Furthermore, development of simulation models can be labor-intensive.

We combine here two prior efforts. We use a version of OpenVZ modified to operate in virtual time [1] with a new parallel network simulator, S3F [2], which was inspired by SSF [3], and RINSE [4]. OpenVZ allows one to run real applications under a real OS and pass messages between simulated and emulated hosts. Users can plug in a real application program rather than be forced to create a simulation model of one. OpenVZ operates in virtual time, not wallclock time, thereby increasing temporal fidelity [1] (unlike most other emulation systems). Freeing the emulation from the real-time clock permits one to run experiments either faster than real time, or slower, depending on the inherent simulation workload. We use S3F for simulating large network scenarios, as it provides sophisticated networking layer protocols and the ability to simulate many many devices such as routers, switches, and hosts creating and receiving background traffic. S3F therefore provides scalability. Coordination of activity between OpenVZ's emulation and S3F's simulation is handled by new extensions to S3F, described in this paper. The current system can run 200+ OpenVZ virtual machines and simulate 100,000+ devices on a single multi-core server.

The contributions of this paper include design of synchronization, event passing and virtual machine control mechanisms in the hybrid system for safe and efficient experiment advancement. We conduct some small-scale experiments to illustrate how changes to timestamps made by the system are bounded, and how these changes behave as a function of the overall simulation load (and are empirically seen to be much smaller than the guaranteed bound). We then conduct large-scale experiments to investigate the scalability of our system, and elements that affect the system performance including number of timelines and synchronization window size.

The remainder of the paper is organized as follows: Section 2 examines the related work. Section 3 describes the system design architecture, while Section 4 illustrates the implementation details of the synchronization mechanism and VE controller of the system. Section 5 evaluates the performance of our testbed in terms of accuracy, speed and scalability, and Section 6 concludes the paper with future work.

2. Related Work

2.1. Network simulation and emulation

Network simulation and emulation are commonly used techniques to test and evaluate networking designs. Representative

network simulators include ns-2 [7], ns-3 [8], SSFNet [9], GTNetS [10], and QualNet [11]. These network simulators generally cannot capture device or hardware characteristics because they do not involve real devices and live networks. On the other hand, the set of commonly used emulation testbeds include EmuLab [12], ModelNet [13], PlanetLab [14], DETER [15], VINI [16], X-Bone [17], and VIOLIN [18]. These emulators present more realistic alternatives to simulators because they combine real physical devices with emulation, but are limited by hardware capacity as they need to run in real time.

Some systems combine both simulation and emulation. One such example is CORE [19]. Recent work by Zheng et al. [1] is similar to CORE in that both of them use OpenVZ to run unmodified code and emulate the network protocol stack through virtualization, and simulate the links that connect them together. A difference is that CORE has no notion of virtual time, while [1] implemented it in their work.

2.2. Virtual time system

Recent efforts have been made to improve temporal accuracy in para-virtualization. DieCast [20] and VAN [21] modify the Xen hypervisor to translate real time into a slowed down virtual time, running at a slower but constant rate. At a sufficient coarse time-scale this makes it appear as though VEs are running concurrently. Our treatment of virtual time differs from DieCast and VAN. The Xen implementations pre-allocate physical resources (e.g. processor time, networks) to guest OSes. In case that the resources have not been fully utilized by guest OSes, the idle VEs (like an operating system would) simply advance the virtual time clock at the same rate as they are busy. By contrast, we advance virtual time discretely, and only when there is an activity in the applications or network.

3. Overall System Architecture

Figure 1 depicts the system design architecture of our system, which integrates the OpenVZ network emulation into a S3F-based network simulator on a single physical machine. The system is capable of running large-scale and high-fidelity network experiments with both emulated and simulated nodes.

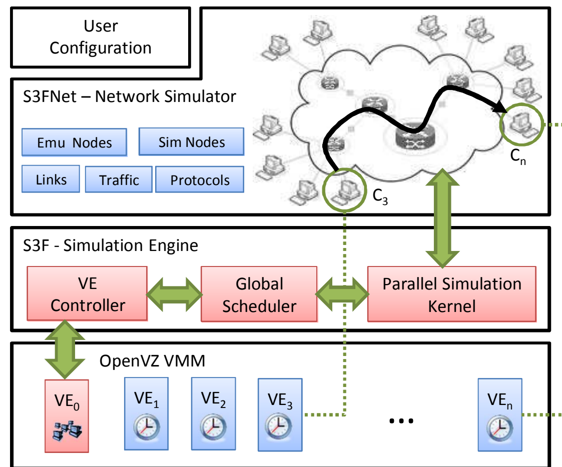

Figure 1 System Design Architecture

The Scalable Simulation Framework (SSF) is an API developed to support modular construction of simulation models, in such a way that potential parallelism can be easily identified and exploited. Following ten years of use, we created a second generation API named S3F [2]. In both SSF and S3F, a simulation is composed of interactions among a number of entity objects. Entities interact by passing events through channel endpoints they own. Channel endpoints are described by InChannels and OutChannels depending on the message direction. Each entity is aligned to a timeline, which hosts an event list and is responsible for advancing all entities aligned to it. Interactions between co-aligned entities need no synchronization other than this event-list. Multiple timelines may run simultaneously to exploit parallelism, but they have to be carefully synchronized to guarantee global causality. The synchronization mechanism is built around explicitly expressed delays across channels whose end-points reside on entities that are not aligned. We call these cross-timeline channels. The synchronization algorithm creates synchronization windows, within which all timelines are safe to advance without being affected by other timelines. More details about S3F are in [2].

In this work, we expand the capacity of S3F by integrating it with the OpenVZ-based network emulation. OpenVZ is an OS level virtualization technology, which enables multiple isolated execution environments (called Virtual Environments (VEs)) within in a single Linux kernel. A VE has its own process tree, file system, and network interfaces with IP addresses, but shares a single instance of the Linux operating system for services such as TCP/IP. Compared with other virtualization technologies such as Xen (para-virtualization) and QEMU (full-virtualization), OpenVZ provides excellent performance and scalability, at the cost of diversity in the underlaying operating system. A VE runs real applications, which interact with emulated I/O devices (e.g. disks), generates and receives real network traffic, passing through real operating system protocol stacks. The only mechanism available to control a VE is the OpenVZ scheduler. When the scheduler frees a VE to execute, the VE runs without interruption or

interaction with any other VE for the period of one "timeslice", a configurable parameter. This presents us with two challenges. One is that the actual length of time the VE runs is somewhat variable, the starting and stopping of that process being handled by the native operating system. In particular, a set of VEs run concurrently will not necessarily receive exactly the same amount of CPU service. This has ramifications for transforming observed real execution durations into virtual time durations. A second challenge is that all interactions between a VE and the network simulator must occur when the VE is not executing. This too has ramifications on assignment of virtual time to message traffic, and on how synchronization is performed.

S3FNet is a network simulator built on top of S3F. Structurally, every VE in the OpenVZ model is represented in the S3FNet model as a host within the modeled network. Within S3FNet, traffic that is generated by a VE emerges from its proxy host inside S3FNet, and, when directed to another VE, is delivered to the recipient's proxy host. The synchronization mechanism needs to know the distinction though between an emulated host (VE-host) or a virtual host (non-VE host), as shown in Figure 1. However, the type of host should make no difference to the simulated passing and receipt of network traffic.

S3F synchronizes its timelines at two levels. At a coarse level, timelines are left to run during an epoch, which terminates either after a specified length of simulation time, or when the global state meets some specified conditions. Between epochs S3F allows a modeler to do computations that affect the global simulation state, without concern for interference by timelines. Good examples of use include periodically recalculating of path loss delays in a wireless simulator, or periodic updating of forwarding tables within routers. States created by these computations are otherwise taken to be constant when the simulation is running. Within an epoch, timelines synchronize with each other using barrier synchronization, each of which establishes the length of the next synchronization window during which timelines may execute concurrently. Synchronization between emulation and simulation is managed by the global scheduler at the end of a synchronization window, when all timelines are blocked. Here it is that events and control information pass between OpenVZ and S3F, using S3F's global scheduler and VE controller. These two core components in the testbed engine designed for integrating emulation and simulation systems will be discussed in details in the next section.

4. Simulation/Emulation Testbed Engine

4.1. Global Simulation/Emulation Scheduler

The global scheduler in S3F is designed for coordinating safe and efficient advancement of the two systems and to make the

emulation integration nearly transparent to S3FNet. Our design forces the OpenVZ emulation to always run ahead of the S3F simulation model, so that VEs operate as traffic sources. Before S3F permits the simulator to advance over a time interval [a, b), we first ensure that all VEs have advanced their own virtual time clocks to at least time b, to ensure that all input traffic that arrives at the simulator with timestamps in [a, b) are obtained first. A packet generated within a VE is given a virtual time stamp based on the VE's clock at the beginning of its timeslice, and the measured execution time until the application code calls the OS to send the packet. The initial send time is as accurate as we can make it. Potentially more parallelism could be exploited if the emulation and simulation executed concurrently. This is a topic we will explore later, as there is sufficient parallelism for the size of problems we're interested in now, and tighter synchrony could paradoxically reduce performance owing to more complex scheduling.

A packet bound for a VE proxy host transits the network model, reaches the proxy host, and is passed to the VE controller, stamped with the arrival time, t. The VE controller delivers the packet to the target VE at the initialization of the first timeslice when the target VE clock is at least as large as t—for a very practical reason. All VEs share the same operating system and its state, and all packets are ultimately obtained by the VE through calls to the operating system; only by extensive modifications to the OS kernel could we build in a per-VE buffering capability that would accept a future packet arrival, and have a VE recognize the packet precisely at its arrival time. We've adopted an approach that is much easier to implement, at the cost of it always being the case that the virtual time at which a packet is recognized (e.g. by a socket read) is larger than the packet's arrival time.

While the synchronization window [a, b) was constructed to ensure that no traffic created within [a, b) is also delivered across timelines within [a, b), it is possible for the VEs to have advanced so far that S3FNet presents a packet to a VE's proxy with a timestamp that is smaller than the VE's clock. This risk seems unavoidable, owing to the coarse grained control we have over VE execution, and when this occurs we deal with it by changing the packet's timestamp.

Integration with the OpenVZ-based emulation brings new features and constraints to the existing synchronization mechanism. Firstly, emulation and simulation never operate concurrently, therefore two clocks actually exist in the system: the current simulation time and the current emulation time; there exist also two types of synchronization window: the emulation synchronization window (ESW) and the simulation synchronization window (SSW). The high level algorithm of the global scheduler is that the system first computes an ESW and runs the emulation for that long, and then injects packets created during that window into the simulator, at the end of the ESW. The new emulation events contribute to the computation of SSW for the next simulation cycle.

Details of the algorithm are described at [22]. Secondly, our system design ensures that the simulation can never run ahead of the current emulation time. Thirdly, once the OpenVZ emulation starts to run, it has to run for at least one timeslice [1], during which no simulation work can interrupt any VE. This system level constraint affects the granularity of the system. Finally, the OpenVZ system introduces opportunities for offering real application specific lookahead for increasing the size of ESW.

4.2. Emulation VE controller

The VE controller is responsible for VE scheduling and message passing between VEs and simulation entities. A given experiment will create a number of guest VEs, each is represented by an emulation host within S3FNet. Each VE has its own virtual clock [1], which is synchronized with the simulation clock in S3F. The VEs' executions are controlled by S3F simulation engine, such that the causal relationship of the whole network scenario can be preserved.

The VE controller uses special APIs to control all guest VEs. It has the following three functionalities. (a) Advance emulation clock: while the VE controller communicates with OpenVZ to start and stop VE executions, it does so under the direction of the S3F global scheduler. Guest VEs are suspended until the VE controller releases them, and they can at most advance by the amount specified by S3F. When guest VEs are suspended, their virtual clocks are stopped and their VE status (e.g. memory, file system) remains unchanged. (b) Transfer packets bidirectionally: the VE controller passes packets between S3FNet and VEs. Packets sent by VEs are passed into S3FNet as simulation inputs and events, while packets are delivered to VEs whenever S3FNet determines they should. By doing so, we provide the notion to the emulation hosts that they are connected to a real network. (c) Provide emulation lookahead: S3F is a parallel discrete event simulator using conservative synchronization [5], and its performance can be significantly improved by making use of lookahead. While S3F may have sufficiently knowledge of the network model state when calculating lookahead, it has no knowledge of the future behavior of an emulation. The VE controller is responsible for providing such emulation lookahead to S3F, the details of which are of course application dependent.

The VE controller's main responsibility is to advance the emulation clock. The VE controller does not drive VEs directly, but allocates timeslices in which to run. A VE is suspended and its virtual clock is paused, except during an allocated timeslice. Once released, a VE runs until the timeslice expires, with its virtual clock increasing as a scaled function of elapsed execution time [1]. Each time the VE controller is invoked by the global scheduler, it is given a window size (ESW) within which all VEs are to advance. Within an ESW, all VEs are independent, i.e. no events from a VE can affect another VE. This independence is either guaranteed by S3F according to channel delays, or derives from the minimum VE scheduling granularity. The VE controller de-

livers packets to VEs just before they begin to execute, and collects generated packets from them after they execute. When the VE controller gets control back after a non-idle VE has run a timeslice, there is variability in the actual length of timeslice the VE consumed, primarily due to the timing resolution of the Linux scheduler. Instead, for a given ESW, whatever length of execution ends up being allocated, the VE controller assumes it is precisely ESW and adjusts the clock accordingly. At the end of a VE controller cycle, the emulation lookahead is calculated and conveyed to S3F through an API. The emulation lookahead is a duration of future virtual time within which a VE will not send packets, so that it will not affect the states of other hosts. In this paper, we demonstrate the promise of emulation lookahead; estimating it and tolerating errors in it is an area of future research.

4.3. Virtual Time Advance

Our modification of the OpenVZ system converts execution time into virtual time; a VE that has advanced in simulation time to t_0 is given T units of execution time, and run. At the end of the execution its clock is advanced to time $t_0 + \alpha*T$, where α is a scaling factor used to model faster ($\alpha < 1$) or slower ($\alpha > 1$) processing. It is important to realize that this is an approximation that treats only at coarse level factors that affect execution time, e.g., caching and pipelining effects. In addition, the scheduling mechanism is not so precise that *exactly* T units of execution time are received, and the VE's actual execution time T' may slightly deviate from T. Nevertheless, in order to keep all VEs in sync with respect to the clock, after execution the VE's virtual clock is explicitly set to $t_0 + \alpha*T$.

In the OpenVZ system, the unit of scheduling (minimum execution time) is a timeslice. We currently set timeslice length TS = 100 μs, but TS is tunable [6]. For the sake of efficiency, the VE does not interact with the VE controller until after its full timeslice has elapsed, at which point packets sent by the VE may be collected, and packets may be delivered to the VE. As we arrange that the emulation always runs ahead of the network simulation, we are assured that each packet arrival lies in the temporal future of the VE-host, and so the packet retains the timestamp received in the emulation.

During the execution, if a message send is performed by the VE, the timestamp on the message is the computed virtual time at which the message leaves the VE to enter the network. In particular, if that departure occurs x units of measured execution time after the beginning of the timeslice, the virtual time of the VE is computed as $t_s = t_0 + \alpha*\min\{x, T\}$, where t_0 is the virtual time at the beginning of the timeslice. The min term is introduced as it is possible for the VE to run longer than T units even though its clock will be advanced only by $\alpha*T$ units, and we need to have virtual time be consistent with that fact. We can bound the amount by which any virtual timestamp is artificially smaller up to $\alpha*T_\varepsilon$, where T_ε denotes the maximum deviation between T' and T. For

the magnitude of TS we have used typically (100 μs), T_ε has tended to be relatively small. It can be up to TS in the worst case, but has proven to be much smaller than that in practice.

At some point the timeline in S3F on which the VE-host is aligned advances its time to recognize the arrival, and normal simulation time advancement techniques deliver the packet to its destination VE-host, say at time t_d. Mechanisms yet to be described ensure that the simulation does not advance farther in time than the VEs have advanced, and so t_d necessarily arrives to a VE with a timestamp smaller than VE's clock. Conceptually, it arrives to the VE later, precisely at the time when the VE begins its next timeslice of execution. In some circumstances this can cause a functional deviation in VE behavior. For example, if the VE in any way "looked" for or blocked on a packet arrival during its previous timeslice at times t_d or greater, it would not see it, and would react to the absence as coded. However, if the VE behavior in the previous timeslice is insensitive to the presence or absence of a packet, the late arrival poses no logical difficulties. When the VE looks for a packet it will find one. From this we see that the effective arrival time of the packet cannot be later than one timeslice TS than its timestamped arrival time.

These observations are summarized more formally below.

Lemma 1: Let t_0 be a VE's clock at the beginning of an execution, and suppose a packet is sent x units of execution time later. The timestamp on the message presented to the network simulator is within $[t_0 + \alpha^*x - \alpha^*TS, t_0 + \alpha^*x]$, where α is the virtual/real time scaling factor and TS is the timeslice length.

Lemma 2: Suppose a packet is delivered to a VE-host at virtual time t_d. That packet is available to the VE no later than time $t_d + \alpha^*TS$.

It is worth pointing out that we cannot construct an end- to-end bound on the error of the packet's timestamp without making some assumptions about how network latencies are different between an arrival at time $t_0 + \alpha^*x$ versus an earlier arrival at time $t_0 + \alpha^*TS$.

5. Evaluation

5.1. Error Analysis

We have seen already that timestamps may be changed, and have bounded the magnitude of those changes. We now examine these changes empirically, using a simple network, which contains two emulation hosts. These two hosts are connected via a link

with 1 Gb/s bandwidth and 100 µs delay. The timeslice T S is also 100 µs, and α is set to 1. During the experiment, a sender application sends constant bit rate (CBR) traffic—meaning the packet inter-arrival time is as constant as virtual time advance can make it—to a receiver application in the other VE. The receiver loops over a blocking socket read, yet has a background computation thread to keep the VE non- idle. For each packet we trace its arrival time at different points along its path, which will reveal where and by how much the virtual time changes. We record the following times: (1) talker: the packet is generated by the sender app (2) vcpull: the sending timestamp presented to S3FNet (3) s3fnet: the delivery timestamp computed by S3FNet (4) vcpush: the packet is delivered and available to the VE (5) listener: the packet is received by the receiver app (6) $vcpush_{err}$: system error, equals to vcpush – s3fnet.

For the sender application, we have tested 25 Mb/s, 100 Mb/s, and 400 Mb/s sending rate. The results are shown in Figure 2 for the 400 Mb/s case. The x-axis indexes the packet, the y-axis shows the times associated with each packet. Slopes decrease with increasing sending rate because inter-packet arrival times decrease.

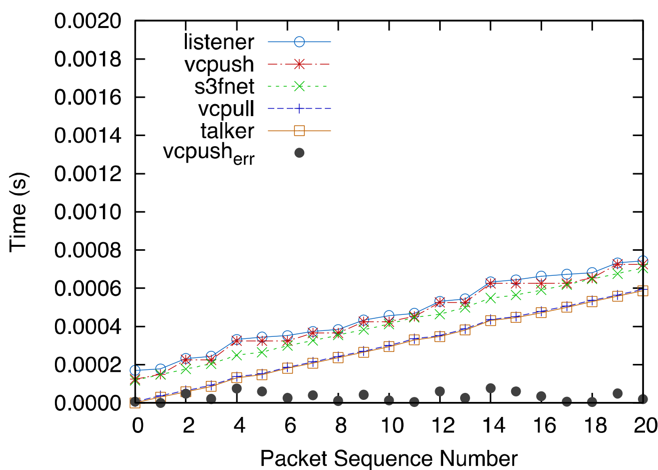

Figure 2 Timestamps during Packets Traverse Route

Although we plot only one sending rate, the behavior of the error in each case is very close, and in this case is bounded by 100 µs—the length of a timeslice. Likewise, the effect of communication latency is the same in each plot, and can be seen in Figure 4. As explained in Section II-C, the sending timestamp we put on a packet is exactly the virtual time when it leaves its VE. There we see clear that talker and vcpull are nearly indistinguishable—the only difference is constant processing delay from application layer to IP layer. The gap between vcpull and s3fnet is the (constant) network latency. Any gap between s3fnet and vcpush is due to the effect described before, that a packet is not pushed to a VE until the VE's clock is at least as large as the packet's arrival time. We know this gap is no larger than α * T S. The data here confirms the theory, and shows

that in this experiment the gap is on average considerably small than one timeslice. The data occasionally shows a gap between vcpush and listener, but this is not caused by our system. Instead, it is caused by multi- task scheduling delay inside the VE, in the same way it exists on a real machine.

The α * T S error bound is an absolute value. When the sender is sending at a very fast rate, e.g. 400 Mb/s as shown, and the inter-packet duration is small, such error and delay approach the inter-packet delay. When the sender is sending at a slower rate, e.g. 100 Mb/s or 25 Mb/s, such error and delay become negligible compared with the relatively large inter-packet delay. We conclude that our system can provide sufficient accuracy for those scenarios that can tolerate these errors. For scenarios that require higher accuracy, one can reduce the length of timeslice, but at the cost of slower execution speed [6].

5.2. Performance Analysis

The testing platform for conducting all the experiments is built on a Dell PowerEdge R720 server with two 8-core processors (2.00GHz per core) and 64 GB RAM, and installed with 64-bit Linux OS. By enabling the hyper-threading functionality, our network simulator can concurrently explore up to 32 logical processors. Given the same size of a network model, i.e. identical topology, and traffic volume and pattern, we would like to investigate means to improve the simulation execution speed without losing accuracy, particularly, performance impact of (1) the number of timelines, and (2) the size of synchronization window.

Number of Timelines: The concept of "timeline" in our testbed is designed for exploiting parallelism of a network model. Simulated nodes, such as hosts, switches, and routers, are partitioned into groups with user-specified rules, such as geographic location or traffic balancing. Each group is aligned to a timeline, and each timeline is assigned to a core during the model execution. Timelines are synchronized through barriers. By increasing the number of timelines, the entire simulation work is distributed to more cores and thus increases the degree of parallelism, at the overhead cost of synchronization. In the first set of experiments, we set up a network with 1024 simulated hosts, among which every two hosts pair up a server-client connection (512 links in total) with 1 Gb/s bandwidth. Each sever is sending constant bit rate traffic flow to its client, and the server-client pairs are evenly distributed among timelines. The above settings minimize the impact of unbalanced simulation workload among multiple timelines with no cross-timeline events, and well-balanced traffic patterns. The inter-packet gap is set to 1 μs to generate sufficient simulation workload. Each experiment is scheduled to run for 1 second in virtual time, and generates 1.54 G events in total. Figure 3 plots the execution time in wall-clock time and event execution rate as we increase the number of timelines.

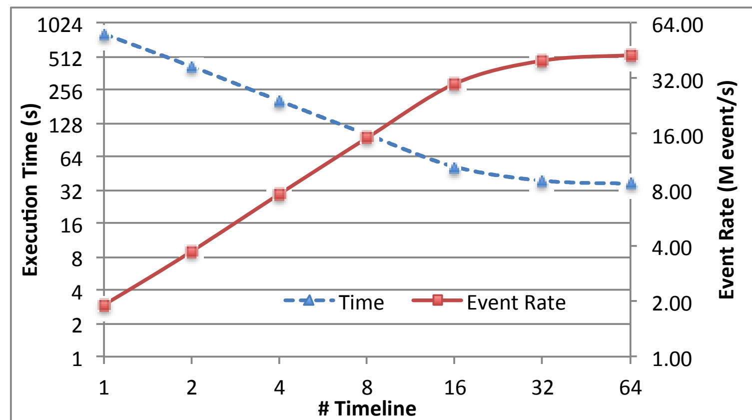

Figure 3 Optimal Simulation Speedup on a Multi-core Architecture Platform

The simulator shows a nearly perfect speedup behavior as we increase the number of timeline up to 16: the execution time continues to drop in half, and the event execution rate continues to be doubled as we double the number of timeline. The improvement slows down when the number of threads grows up to 32 due to the heavy involvement of the hyper-threading technique. We have conducted the same set of experiments again with hyper-threading turned off, and the comparison shows that using hyper-threading can achieve a 1.36 speedup factor for our simulator when the number of timelines is greater than or equal to 32. The results show that given sufficient workload within each synchronization window, our simulator can maximize the hardware parallelism capabilities by creating the same number of timelines (i.e. software threads) as the number of available logical processors, with an event rate as high as 40 million-events/second. Further increasing the number of timeline gives us little performance gain. The above experimental results profile the best achievable speedup performance of our simulator by maximizing the work in the parallel section of our simulator. The rest overhead is due to OS-dependent control elements, such as multi-thread scheduling and inter-processor management that are beyond our control.

Synchronization Window: Synchronization windows indicate how long the emulation or the simulation can proceed without affecting entities on other timelines. The length of the windows are computed at synchronization barriers based on the detailed network-level and application-level information, such as minimum link delay and minimum packet transfer time along the communication paths, or network idle time contributed by the simulated devices that not actively initiate events (e.g. server, router, switch), or from the lookahead offered by the OpenVZ emulation. A detailed algorithm description is presented in [22]. In this set of experiments, we investigate the performance impact of synchronization window on our simulation/emulation testbed. We set up a network with 64 *emulated* hosts, among which every two hosts pair up a server-client connection, and 32 links in total. Each sever sends constant bit rate UDP traffic flow with constant 1500 Byte packet size to its client. The emulation time slice is set to 1

ms, and the networking environment created in S3FNet has 1 Gb/s bandwidth and 1 ms link delay. We vary the sending rate with 100 Kb/s, 1 Mb/s and 10 Mb/s in the above scenarios and record both the emulation time and the simulation time every 10,000 packets. We then pre-calculate a constant lookahead based on the observed average inter-packet gap, essentially creating larger synchronization windows, and re-run the experiments for comparison. We observe the nearly identical sending/receiving traffic patterns for each scenario with and without lookahead, which indicates little experiment accuracy loss with the presence of the lookahead. Figure **4** compares the execution times for both emulation and simulation.

With emulation lookahead, the execution time, both emulation and simulation, significantly reduces for 100Kb/s and 1 Mb/s sending rate. Here is the explanation: for 1 Mb/s sending rate, given 1500 byte packet size, the average inter-packet time is around 12 ms, which equals to 12 timeslices; and accurate emulation lookahead should predict that amount — promise that a VE will not generate any events within the next such amount of time, and offer this value to S3F for computing the next emulation synchronization window (ESW). The new ESW increases to approximately 12 timeslices in length and thus minimizes the emulation overhead. Since the emulation is now running far ahead of time as compared with the case without emulation lookahead, and no events are injected into the simulation's event lists, the simulation also takes advantage of the empty event lists to compute a large simulation synchronization window. Therefore, the execution times on both simulation and emulation are significantly reduced, and same is true for 100 Kb/s case. However, little improvement is observed for the 10 Mb/s case, because ESW generated by emulation lookaheads (1.2 ms) is close to one time slice. Another observation is that by offering accurate lookahead, the execution times for transmitting 10,000 packets with all the sending rates are very close. This is actually one benefit with our virtual-time-embedded emulation system that we can accelerate low traffic load emulation experiments by utilizing available system processing resources.

Providing good simulation lookahead is always challenging for conservative network simulation, and our virtual-time-based simulation/emulation testbed gives another opportunity for providing good emulation lookahead. The above experimental results clearly indicate the huge performance gain that a good lookahead mechanism can bring to our system, and thus strongly motivate our ongoing work of investigating emulation lookahead from source code/binary analysis.

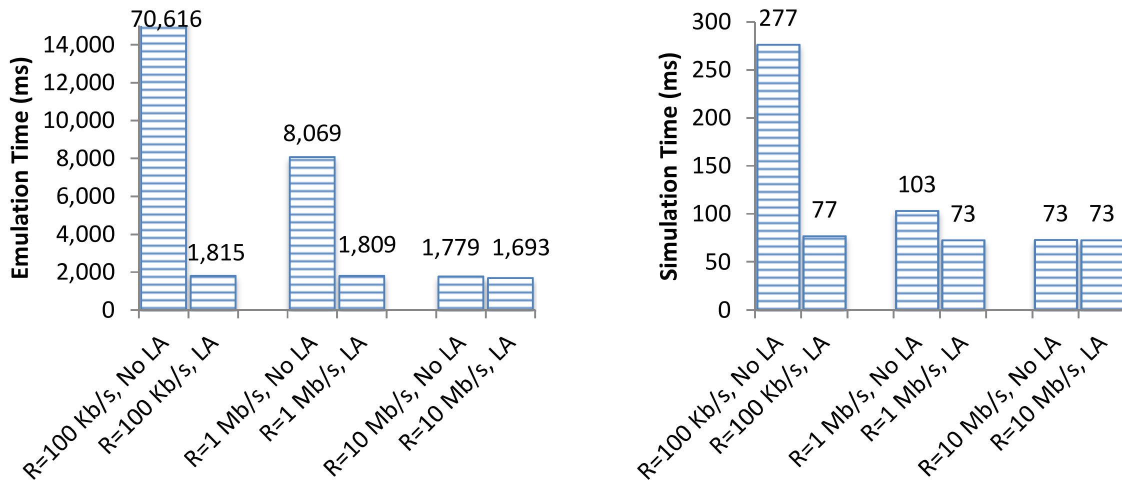

Figure 4 Execution Time Comparison with Lookahead (R – Sending Rate, LA – Lookahead)

5.3. Scalability Analysis

Our testbed was used to investigate a distributed denial of service (DDoS) attack in an advanced metering infrastructure (AMI) established by 448 smart meters in a typical 4 by 4 block neighborhood [22]. Here, we increase the size of the model by adding the number of neighborhoods (hence number of simulated meters and links, and length of communication paths). There is one egress point in each test case, to which every meter periodically sends data traffic. Each experiment uses 32 timelines and runs for 1 hour in virtual time. Table 1Table 1 summarizes the experimental results. We are able to simulate a system as large as 64 by 64 neighborhood, with 100,000 + meters, 600,000+ links and 437 billion of events. As the model size increases, the event rate remains around 6M ~ 7M event/second. The results indicate that our testbed's performance scales.

Table 1 Simulation Scalability Test Results Using AMI DDoS Test Cases

Block	#Host	#Link	Simulation Time (s)	#Events (M)	Event Rate (M Event/s)
2 X 2	112	392	13	62.3	4.78
4 X 4	448	1,953	69	509.9	7.34
8 X 8	1,792	8,691	658	4,414.0	6.70
16 X 16	7,168	36,501	3,958	25,636.3	6.48
32 X 32	28,672	149,702	16,491	107,855.7	6.54
64 X 64	114,688	604,566	63,180	437,452.4	6.92

6. Conclusions and Future Work

In this paper we present a system that integrates an OpenVZ-based network emulation system [1] into the S3F simulation framework [2]. The emulation allows native Linux applications to run inside the system; and the emulation is based on virtual time, which both provides temporal fidelity and facilitates the integration with the simulation system. We design and study the global synchronization and VE controlling mechanism in the system. The current system requires both OpenVZ and S3F to reside on the same shared memory multiprocessor. Our future work includes separating these to support multiple machines running virtual machine managers, not necessarily all running the same virtual system. We are also interested in means of estimating lookahead from within the emulation and providing it to the simulation, to accelerate performance.

Acknowledgements

This material is based upon work supported in part by the Department of Energy under Award Number DE-OE0000097, by the Boeing Corporation, and the International Cooperative R & D Project of the Korea Institute of Energy Technology Evaluation and Planning (KETEP) grant funded by the Korea government Ministry of Knowledge Economy (20111030100020).

References

[1] Y. Zheng and D. Nicol, “A virtual time system for openvz-based network emulations,” In Proceedings of the 25th Workshop on Principles of Advanced and Distributed Simulation (PADS). 2011, pp. 1–10.

[2] D. Nicol, D. Jin, and Y. Zheng. “S3F: The Scalable Simulation Framework revisited,” In Proceedings of the 2011 Winter Simulation Conference (WSC), Phoenix, AZ, December 2011, pp. 3283–3294.

[3] D. Nicol, J. Liu, M. Liljenstam, and G. Yan, “Simulation of large scale networks using ssf,” In Proceedings of the 2003 Winter Simulation Conference, 2003., vol. 1. IEEE, 2003, pp. 650–657.

[4] M. Liljenstam, J. Liu, D. Nicol, Y. Yuan, G. Yan, and C. Grier, “Rinse: the real-time immersive network simulation environment for network security exercises,” in Proceedings of Workshop on Principles of Advanced and Distributed Simulation, 2005. IEEE, 2005, pp. 119–128.

[5] R. Fujimoto, “Parallel discrete event simulation,” in Proceedings of the 21st Winter Simulation Conference. ACM, 1989, pp. 19–28.

[6] Y. Zheng, D. Nicol, D. Jin, and N. Tanaka. “A Virtual Time System for Virtualization-Based Network Emulations and Simulations,” Journal of Simulation, vol. 6, no. 3, pp. 205–213, August 2012.

[7] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu et al., “Advances in network simulation,” Computer, vol. 33, no. 5, pp. 59–67, 2002.

[8] The ns-3 project. http://www.nsnam.org.

[9] J. Cowie, D. Nicol, and A. Ogielski, “Modeling the global internet,” Computing in Science & Engineering, vol. 1, no. 1, pp. 42–50, 2002.

[10] Gtnets, http://www.ece.gatech.edu/research/labs/maniacs/gtnets.

[11] Scalable network technologies. http://scalable-networks.com.

[12] Emulab, http://www.emulab.net.

[13] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kosti´c, J. Chase, and D. Becker, “Scalability and accuracy in a large-scale network emulator,” ACM SIGOPS Operating Systems Review, vol. 36, no. SI, pp. 271–284, 2002.

[14] Planetlab, http://www.planet-lab.org.

[15] T. Benzel, R. Braden, D. Kim, C. Neuman, A. Joseph, K. Sklower, R. Ostrenga, and S. Schwab, “Experience with DETER: A testbed for security research,” In Proceedings of the 2nd International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities, 2006, pp. 10–388.

[16] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford, “In VINI veritas: realistic and controlled network experimentation,” In Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications. ACM, 2006, pp. 3–14.

[17] J. Touch, “Dynamic Internet overlay deployment and management using the X-Bone* 1,” Computer Networks, vol. 36, no. 2-3, pp. 117–135, 2001.

[18] X. Jiang and D. Xu, “Violin: Virtual internetworking on overlay infrastructure,” In Proceedings of Parallel and Distributed Processing and Applications, pp.937–946, 2005.

[19] J. Ahrenholz, C. Danilov, T. R. Henderson, and J. H. Kim, “CORE: A real-time network emulator,” In Proceedings of Military Communications Conference, IEEE, 2008, pp. 1–7.

[20] D. Gupta, K. V. Vishwanath, and A. Vahdat, “DieCast: testing distributed systems with an accurate scale model,” in Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, ser. NSDI’08. Berkeley, CA, USA: USENIX Association, 2008, pp. 407–422.

[21] P. K. Biswas, C. Serban, A. Poylisher, J. Lee, S.-C. Mau, R. Chadha, C.-Y. J. Chiang, R. Orlando, and K. Jakubowski, “An

integrated testbed for Virtual Ad Hoc Networks," In Proceedings of the International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities, , vol. 0, pp. 1–10, 2009.

[22] D. Jin, Y. Zheng, H. Zhu, D. M. Nicol, and L. Winterrowd. "Virtual Time Integration of Emulation and Parallel Simulation," In Proceedings of the 26th Workshop on Principles of Advanced and Distributed Simulation (PADS), Zhangjiajie, China, 2012, pp.120–130.