Open Network Emulator: A Parallel Direct Code Execution Network Simulator

Vedavyas Duggirala and Srinidhi Varadarajan
Department of Computer Science
Virginia Tech
Blacksburg, VA
{vduggira, srinidhi}@cs.vt.edu

Abstract—In this paper, we present the Open Network Emulator (ONE) a network simulator that combines the controllability and scalability of simulation with the direct code execution properties of emulation and experimental testbeds. ONE has two novel features. First is a compiler framework that automatically transforms existing network application/protocols written in imperative languages such as C and C++ into composable modules, which can then be combined to create arbitrarily complex network stacks. This compiler framework obviates the need for heavy-weight virtualization and enables ONE to execute multiple virtual hosts, each with its own application and network protocol stack, within a single process. The second novel feature of ONE is a new time model called Relativistic Time that combines the controllability of virtual time with the fidelity of real-time. To implement ONE, we ported the complete TCP/IP stack from within the Linux kernel (including the sockets interface, TCP, UDP IPv4 and v6, ICMP, IGMP, traffic shaping, netfilters, routing and ARP) and well-known configuration and packet tracing applications such as ifconfig and tcpdump. Existing network applications can be compiled and instantiated within ONE without requiring any source code change. We validated the fidelity of ONE by comparing the packet arrival times of multiple traffic generators on a real network against the arrival times with ONE emulation. Our preliminary performance evaluation of ONE on an 8 core system shows that ONE is highly efficient and can run over 450 virtual hosts connected over point-to-point gigabit links, while still retaining linear behavior.

Keywords: Network Simulation, Network Emulation, Parallel Discrete Event Simulation, Time dilation

I. INTRODUCTION

As the Internet continues to evolve, the ubiquitous availability of broadband and mobile connectivity is enabling the rapid proliferation of new protocols, applications and services that rely on the “always on” nature of the network. Concomitantly, the increasing sophistication of mobile devices such as smartphones and tablet computers combined with backend computational resources in the cloud has created a new class of highly distributed applications that seamlessly integrate multiple information sources. Developing correct applications and protocols in this environment requires large-scale test environments that are representative of the scale of the Internet and yet have the controllability and fidelity needed for verification and validation of emerging network services.

Historically, simulation, emulation and large-scale controlled testbeds have been used for protocol validation. While simulators such as ns-2[1] and OPNET[2] offer an efficient execution model, they require that the protocol under test be written in their event driven model. The simulated protocol is then refined and converted to a real-world code implementation usually written using imperative programming. However, there is no easy way to ensure the equivalence of the simulated protocol and its real world code version leading to verification and validation problems. A secondary issue in network simulators stems from its clean-room implementation. Since the TCP/IP protocol stack in simulators is generally written from scratch, it does not exactly emulate real-world protocol stacks, in particular the idiosyncrasies of real-world TCP/IP, which can significantly affect performance [3]. Even when simulation models have been empirically validated against an implementation, over time, the implementation is iteratively refined, leading to significant divergence from the model. In spite of these deficiencies, simulation remains the primary tool of choice since studies are scalable, controllable and highly repeatable.

Large scale experimental testbeds like Planetlab [4] can directly run real-world implementations, thus mitigating the verification and validation problem. By using actual network components, they offer a degree of realism that can’t be matched by simulators. However, they are expensive to scale and studies are often orders of magnitude smaller than comparable simulation experiments. Furthermore, experimental testbeds operating in real-time lack controllability and repeatability. Testbeds are often shared resources and it might not always be possible to isolate the effects of other experiments running concurrently. Finally, experimental testbeds are limited by currently available state-of-art, which precludes their use in studying the effects of emerging or in-development network technologies that do not yet exist. Given the significant system interactions between processor, memory hierarchy and the network interconnect, one cannot easily simply run an application on the 1 Gbps network available in a testbed.
and predict its performance on an emerging 100 Gbps interconnect.

Emulators like Emulab[5], ModelNet[6], Dummynet[7] and NistNet[8] can directly execute real-world network applications, but the network itself is modeled using traffic shaping queues to provide the configured delay, latency and bandwidth. Operating system scheduler’s overhead, jitter in timer interrupts and processor oversubscription cause deviations in delay from that intended by the traffic shaper. This noise is the main source of verification and validation errors in emulators resulting in repeatability and controllability issues similar to experimental testbeds.

In this paper, we present the ONE network simulator that combines the controllability and scalability of simulation with the direct code execution properties of emulation and experimental testbeds. ONE has two novel features. First is a compiler framework that automatically transforms existing network application/protocols written in imperative languages such as C and C++ into composable modules, which can then be combined to create arbitrarily complex network stacks. This compiler framework obviates the need for heavyweight virtualization and enables ONE to execute multiple virtual hosts, each with its own application and network protocol stack, within a single process. The second novel feature of ONE is a new time model called Relativistic Time that combines the controllability of virtual time with the fidelity of real-time. To implement ONE, we ported the complete TCP/IP stack from within the Linux kernel (including the sockets interface, TCP, UDP IPv4 and v6, ICMP, IGMP, traffic shaping, netfilters, routing and ARP) and well-known configuration and packet tracing applications such as ifconfig and tcpdump. Existing network applications can be compiled and instantiated within ONE without requiring any source code change. We validated the fidelity of ONE by comparing the packet arrival times of multiple traffic generators on a real network against the arrival times within ONE emulation. Our preliminary performance evaluation of ONE on an 8 core system shows that ONE is highly efficient and can run over 450 virtual hosts connected over point-to-point gigabit links, while still retaining linear behavior.

The rest of the paper is organized as follows. ONE compiler framework is described in Section II. We describe the Relativistic Time(RT) model and its performance implications in Section III. Section IV gives implementation details of ONE. In Section V we present performance evaluation results and in Section VI we discuss related work.

II. THE ONE COMPILER FRAMEWORK
To operate within the confines of the direct code execution paradigm, wherein the network application/protocol under test requires no modifications, necessitates a new composition paradigm. We will illustrate this need by a series of examples.

Let us start with the case depicted in Figure 1. Here, two telnet applications are linked against a single IP stack, emulating one virtual end-host.

![Figure 1: A simple compositional model with 2 telnet applications linked with a single IP stack. The composition creates a single virtual host.](image)

A. Process per Virtual Node
In order to model the composition shown in Figure 1, we start with a simple approach, which we call the process per virtual node model as shown in Figure 2(a). This model allows a single network application – say telnet – to be linked to an IP stack. The external references of the telnet application will be bound to the stack, which then uses an underlying discrete event simulation layer. Multiple such processes can be run on a workstation to create a virtual network. This approach has several limitations. Since the IP stack shown in Figure 1 is shared between all applications running at that node, traffic from the various applications would be influencing each other through competition for buffers in the IP stack. This effect of competing flows on each other cannot be studied in the simple process per virtual node model presented in Figure 2(a), since each telnet would have an independent IP stack. While it is possible to synchronize the state of the two IP stacks (shown in Figure 2(a)) using techniques like IPC mechanisms, it would be highly intrusive on the IP stack implementation leading to correctness and scalability issues.

B. Thread per Virtual Node
To address the above concerns, we consider another model based on threads as shown in Figure 2(b). In the threads model, the composition shown in Figure 1 can be achieved by modifying the telnet application to create two threads, or inserting a piece of stub code that creates two threads, with the start function of each thread set to the entry point - main() - of the telnet application. As before, the emulated IP stack linked to the application will resolve external references to the IP layer.

The major problem with this approach arises from updates to global variables. In particular, network applications such as telnet are not necessarily thread safe. Since threads share...
global variables, a telnet thread modifying a global variable will inadvertently change the state of the other – unrelated - telnet thread causing erroneous behavior.

Ideally, we need 2 copies of all the global variables used in the telnet application. In programs that are explicitly threaded in design, sharing of global state is intentional. In our case, this sharing is neither intentional nor necessarily desirable. On the other hand, to create a shared IP stack, we need to share global state within the IP stack between the threads running through the IP stack. We also need to serialize access to global state through the use of mutex locks, which involves significant modifications to the source code. Even if we could make these modifications, we cannot ensure that the resultant code is deadlock free. In effect, we are back to our original verification and validation problem faced in network simulation. There is no easy way to formally ensure that the threaded version of the network application is equivalent to the unthreaded version.

The threads example illustrates the crux of our problem – conflicting needs – the need to avoid sharing global state between threads of the telnet application and the need for sharing global state between the threads running through the IP stack. To summarize, we need a composition model that allows arbitrary sharing of global state. Such a model can subsume both the thread and process models, since it can allow both complete sharing of global state as in threads as well as no sharing as in processes.

The design goal of the ONE, shown in Figure 3 is to support multiple virtual hosts (a router, switching device or an end-host), each with a protocol stack and multiple applications, within a single process.

An additional advantage of this design is that since all the virtual hosts are in same process, memory overhead of inter-host packet transmissions can be optimized.

C. ONE Composition Model

ONE’s composition system that enables the selective sharing was described previously in [9]. Traditionally, one or more namespaces map to a single address space. The core idea behind the Weaves model is to map a single namespace to multiple address spaces. Hence, for example, a single global variable x would map to different addresses depending on the desired composition. By performing the mapping of namespace to address space at runtime, Weaves enables arbitrary sharing of state, thereby subsuming both the process and thread models.

In the Weaves model, a collection of one or more object files is compiled and then instrumented through binary instrumentation to produce a module. Using the example above, the source files comprising the TCP/IP stack would be compiled into a single IP stack module. The instrumentation identifies global variables accessed within a module and indirects their access to point to a distinct address space that is created at runtime. The address space in the Weaves model – called a Weave – consists of the data contexts of instantiations of modules. Each instantiation of a module is identified by a unique identifier and a Weave describes a tuple of such unique identifiers. To instantiate the example of Figure 1, the weaves model creates distinct address spaces (weaves) for (a) <telnet 1, ip stack 1>, (b) <ftp 1, ip stack 1>, (c) <telnet 2, ip stack 1> and (d) <telnet 3, ip stack 2>. Since the instantiation ip stack 1 is common to the first three address spaces, the global variables within the IP stack are shared across the three address spaces. Since telnet 1 and telnet 2 represent distinct instantiations, their address spaces are separated into two distinct weaves, thereby separating their global state. The Weaves model also introduces the notion of a control flow called a string, which operates within a single address space. Conceptually, a string is similar to a thread, however, unlike threads which share their namespace and address space, a string operates within a distinct address space defined by the weave. In the example shown in Figure 3, there are four distinct strings running in each of the four distinct address spaces listed above. By controlling the mapping between a namespace
and an address space, the Weaves model can selectively separate or share global state thereby producing compositions with arbitrary state sharing properties. Weaves[9] uses ELF [10] binary instrumentation in its implementation. This instrumentation model is not portable and currently the implementation supports 32 bit x86 platforms. The 32 bit limitation restricts the amount of memory that can be used in the simulation to 4GB, which significantly hinders scalability. Furthermore, Weaves binary instrumentation technique, which works by indirecting Global Offset Table (GOT) entries, does not apply to 64 bit x86 platforms, which uses rip register relative addressing for position-independent code (PIC) and hence could not be used as the basis for our work.

These hurdles led us to develop our composition model similar to Weaves that operates without non-portable binary instrumentation, while still being transparent to the source application. Our goal was to ensure that the composition model was scalable, portable and easy to deploy even in the context of other simulators such as ns3, which are developing direct code execution capabilities and hence face the same selective sharing issues. To achieve a selective state sharing solution which is an application and machine architecture independent we developed the ONE compiler framework.

ONE operates by instrumenting an intermediate code representation of the application source code, which is target architecture neutral rather than relying on processor specific machine code instrumentation. In the ONE model, application modules are compiled (without modification) using the LLVM gcc compiler into the LLVM intermediate representation (IR). The LLVM project [11] provides a set of tools to implement the compiler tool chain. LLVM uses a SSA (single static assignment) based IR format as target of high level language front-ends. It provides optimization passes on the IR and backends for code generator from IR to major platforms including x86 32 bit and 64 bit architectures. The IR format is simple and powerful, capable of expressing high level language constructs like type, storage class of operands and debug metadata. By choosing to instrument at the IR level, ONE’s composition model is both source language as well as target architecture neutral, enabling its use in a variety of projects.

D. IR Transformations

The IR format allows us to perform many operations equivalent to high level language transformations and instrumentation at the language independent IR level. In the IR format all the operands (except constants) are accessed via their addresses.

To indirect accesses to global state, ONE’s IR instrumentation inserts a function call (get_target_addr) provided by our runtime before every access of a global variable and replaces the global variable with the return value of the function call. Figure 4 shows the transformation performed by ONE on the IR.

```
int global_var;
void foo() {
    int *addr;
    addr = get_target_addr(wv_id, &global_var);
    *(addr)++;
}
```

E. Selective Sharing Runtime

ONE runtime allocates independent copies of global state for each object and builds a lookup table from a configuration file, which specifies the state sharing relationships between objects. The instrumentation function get_target_addr uses the lookup table to resolve global variables to the right address in each thread of execution.

III. RELATIVISTIC TIME

In this section, we describe example network experiment scenarios that illustrate the limitations of time models commonly used by simulators and emulators and the need for our Relativistic Time (RT) model.
A. Limitations of Simulation and Emulation

Consider the code snippet shown in Figure 5 of a simplified server connected to a client. In simulation, the time advances only at the end of event processing calls recv_message and send_message. The processing of an event itself is assumed to take zero time. Simulation time cannot account for the time spent processing a message, when executing real-world applications. Simulation runs in virtual time, whose progress is not continuous but discrete. Virtual time can be frozen or set to any arbitrary monotonically increasing value. In addition, the rate of progress of virtual time is dependent on the type of the event and event density at the host.

Now consider emulating a client and server running on two distinct emulator nodes connected by a 100 Mbps link and the network experiment is trying to simulate a 1 Gbps link. Emulation would correctly account for the time spent in processing a message, but send_message and recv_message cannot correctly account the time taken by the virtual link with higher capacity than the physical link.

B. Network Scaling

To correctly emulate the above scenario where emulated capacity is greater than physical link capacity, the emulator should be able to control the notion of time at both end hosts. We define scale factor (SF) as the ratio of bandwidth of physical link and emulated link (\(SF = B_P / B_E\)). The emulator should freeze the clock before every receive message and update it by SF times the real time it took to transmit the message (in this example, ten times less than wallclock time taken by the emulator).

In this way, virtual time in an emulator allows the simulation of higher capacities, though at increased real time.

An obvious downside of the above simple solution is that it does not scale well, when there are multiple such end hosts with varying link capacities. Further, the same server might be connected to different clients with varying link speeds. Updates to time at end hosts with different links have to be applied in correct order and deal with messages in transit. The parallel discrete event simulation (PDES) community has developed various time synchronization algorithms to provide a globally consistent notion of time i.e., Global Virtual Time (GVT) across all the nodes. We discuss the advantages of our RT over PDES in Section III.F

Virtual time in emulators is useful not just in cases where there is resource over subscription, but for resource under subscription too. Consider the same scenario, with simulated link capacity of 10 Mbps, which the emulator can easily run in real time. Due to OS timer interrupt frequency and influence of background traffic, a packet intended for time \(t\) would arrive at \(t \pm \delta t\). The emulated traffic is in a saw-tooth pattern, instead of the original [12]. This is the root cause of non-reproducibility of emulation. In a system with virtual time, this would not be an issue since, irrespective of when the packet arrives in real time, the virtual time is set to \(t\).

![Figure 6. Capacity mismatch between virtual network and emulator](image)

C. CPU Scaling

Consider the scenario in Figure 6, where two emulator nodes are connected by a 100 Mbps link. We would like to simulate two clients talking to a server over two independent 10 Mbps point-to-point links. Further, suppose the CPU utilization of the clients is negligible. Our two node emulator is incapable of simulating this scenario, though it has the required processing power and bandwidth, because it runs in real time. We would need at least three physical nodes to emulate it. If we have a notion of GVT, we can multiplex the two clients on one physical node. Each client would have its own clock, which advances only during its execution.

As the above discussion illustrates, emulators are unable to support many of the interesting network experiment scenarios, in spite of having sufficient resources, due to lack of global virtual time.

D. Time Dilation and Time Warp

Relativistic Time (RT) is based on the observation that time is just a monotonically increasing ordinal value. By controlling (overloading) the time querying functions of network applications, we can hide the amount of wallclock time that elapsed between two instants of virtual time. By varying the relative rate of progress on virtual time \(w.r.t\) real time, the simulator can thus handle both resource over-subscription and under-subscription. We define two operations, dilation and warp on virtual time which alter its rate of progress

a) Time Dilation

The dilation operation slows the progress of virtual time \(w.r.t\) real time. Dilation is used to handle resource over subscription. Computational over subscription is the ratio of simulated nodes (\(N_s\)) to number of physical nodes (\(N_p\)) in the emulator. Bandwidth over subscription is the ratio of cross-sectional bandwidth of the emulated network (\(B_E\)) and

\[\text{Bandwidth over subscription} = \frac{B_E}{B_P}\]

\[\text{Computational over subscription} = \frac{N_s}{N_p}\]

105
the physical network \((B_p)\). The minimum dilation necessary to correctly simulate such a scenario is
\[
\text{Scaling factor} = \text{Max } \left( \frac{N_s}{N_p}, \frac{B_s}{B_p} \right)
\]

b) Time Warp
The warp operation speeds up the progress of real time w.r.t. virtual time. When all the nodes in an emulation are blocked waiting for next event, the emulator can jump to the next event in the simulation. The warp operation compensates for resource under subscription.

E. Host Time Ahead of Relativistic Time
In network simulation, the computational load is not always uniformly distributed across all the nodes. Some subset of nodes, either clients or servers or routers have greater computation load. In such cases, we allow the host to get ahead of GRT as long as the host does not perform a receive operation (e.g. \texttt{receive_message}, \texttt{select}, \texttt{poll}). When it performs receive operation the host is blocked, till GRT catches up to the host’s time. This is because other hosts that could send a message to it are behind in time and have not had a chance to run.

F. Performance Implications
Conservative PDES \cite{13} algorithms have traditionally been used in network simulation. Lookahead used by these algorithms is based on link characteristics. In practice the actual lookahead is dynamic and varies during the course of simulation depending on changes in topology, state of network protocol, and traffic patterns. These algorithms send \texttt{null} messages to compute current lookahead and hence determine which events are safe to execute. Scaling factor in RT uses worst case lookahead, based on static link characteristics. When dynamic lookahead is the same as static lookahead, RT processes the same set of events as a conservative algorithm would. Though, unlike conservative algorithms, RT does not exchange any \texttt{null} messages.

When the dynamic lookahead is greater, all the nodes in the simulation would be blocked waiting on the next event and RT would perform a warp operation and speedup the simulation. RT can thus take advantage of these dynamic changes in lookahead without explicitly calculating lookahead through \texttt{null} messages.

To achieve good lookahead, simulation models need to be (re-)written to provide lookahead hints \cite{14}. While it might be possible in network models, which are based on simple queues, it is very complicated when real implementations are used. RT’s ability to track dynamic changes in lookahead is a major advantage over conservative algorithms.

IV. ONE IMPLEMENTATION

A. User Level TCP/IP Stack
To support our goal of running unmodified socket API based network applications, ONE uses a full-fledged TCP/IP stack derived from Linux kernel version 2.6.18. We provide dummy abstractions of the rest of the linux kernel (scheduler, processes, files system, etc) to compile the networking subsystem as a user-level shared library. The original kernel source is otherwise unmodified.

The shared library exposes the POSIX Socket API to applications. The library has \texttt{send_msg}, \texttt{receive_msg}, and timer function calls to interface with a simulator. Using the timer function, the simulator can
control the notion of time seen by the network stack and applications linked against it.

The network stack uses a special thread called *interrupt thread* to provide a network hardware device abstraction. The interrupt thread is used to offload message transmission and reception from the simulator. Interrupt enabling and disabling calls (*cli* and *sti*) in the kernel code, are used to serialize any accesses to the network stack’s internal data structures by the interrupt thread and application threads linked against the shared library. The interrupt thread queues up events, if the interrupts are disabled and processes them in order, when they are re-enabled.

Common network configuration utilities like *ifconfig* and *route* are linked against the shared library to configure the network stack. The network stack can also dump packet traces in the standard libpcap format.

B. Calendar Queue

ONE uses a calendar queue [15] to represent the pending event list of the simulation. We use coarse-grain locking for the calendar queue. This simplifies the design, when the calendar queue is periodically resized for optimal performance. We are exploring alternative fine-grained locking strategies to reduce lock contention.

ONE uses the GNU Scientific Library for random number generation.

ONE stops the host and GRT clocks during enqueue and dequeue event operations. The emulated application sees these operations as happening instantaneously in virtual time. This ensures that the temporal fidelity of hosts is not affected by simulator overhead. Similar technique could be used to isolate I/O subsystem overhead, but it is beyond the scope of this paper.

We use processor affinity to fix virtual hosts to assigned processors. Otherwise, we see degraded performance due to excessive process migration by the OS scheduler trying to load balance. An optimal positioning would depend on the topology and traffic patterns in virtual network. We feel this problem needs to be further studied and automatic assignment of processor affinities should be made.

V. EXPERIMENTAL EVALUATION

Our experimental evaluation is performed on a single node with two 2.8 GHz Intel Xeon E5462 Quad core processors (total 8 cores) with 8 GB of RAM.

Our design choice of virtualizing only the network stack is based on our goal of running as many virtual hosts as possible. To the best of our knowledge, the largest experiment with virtual machines (VM) has been 320 OpenVZ instances on a machine with 16 GB RAM [16]. ONE is able to emulate 450 network instances using only half that memory. ONE can easily emulate three times the number of network hosts as the most efficient VM based emulators. Note that the maximum number of virtual hosts is also limited by the amount of memory required by the applications linked against the virtual hosts.

Our experiment consists of 225 infinite source and infinite sink pairs (total 450 nodes) exchanging CBR traffic, connected by a 1 Gbps, 4 us latency link. We choose a high bandwidth, low latency link to demonstrate that our simulator can handle heavy oversubscription.

Figure 8 shows the average application throughput achieved (963 Mbps for TCP and 924 Mbps for UDP). We find little variation in bandwidth in spite of the emulation being highly oversubscribed. This is because applications run under virtual time.

![Figure 8. Bandwidth vs Number of Virtual Hosts](image)

Figure 9 shows the runtime of emulation vs. the number of nodes. Each source-sink pair is exchanging 10000 packets of 512 bytes each. TCP’s sliding window algorithm restricts the number of packets enqueued by the simulation, while UDP has no such limit. Consequently, the size of the pending event list is higher for UDP, and enqueue and dequeue event operations take longer. This accounts for the higher runtime of UDP compared to TCP.

![Figure 9. Runtime vs Number of Nodes](image)
Figure 10 shows the number of events processed by our simulator. Our simulator has four types of events: timer, send, receive, and block events. Unlike a simulator, where inter model transmissions within a virtual host appear as events in the pending event set, in our simulation only across the virtual host message transmission count as event. Every message transmission counts as two events (send and receive). The send event is used to free the transmit buffers on the sender side.

Figure 11 shows the runtime for 200 pairs (400 virtual hosts) when the number of packets in the emulation is increased. As shown in Figure 10 and 8 our simulator scales linearly both wrt number of virtual hosts and number of packets being transmitted inspite of being heavily oversubscribed.

Figure 12 shows the packet arrivals at a sink for the first 25 packets. The source is sending pareto traffic. The lower band shows that the packet arrival time in simulator does not vary with load (increased number of flows). It is well known that in emulators [12] the emulated link characteristics are distorted by background traffic, leading to validation and reproducibility issues. Our simulator running in virtual time has no such issues.

VI. RELATED WORK

We classify the existing techniques broadly into three categories simulators, emulators and parallel discrete event systems.

A. Simulators

ns-2 [1] has been the most widely used network simulator in networking research. ns-2 is a pure simulator and cannot run real-world network applications. ns-3 is a non-backwards compatible ground up rewrite of ns-2. Ability to link against socket based codes was one of the original design goals [17] of ns-3. While, it has an optional full-fledged real-world implementation of TCP/IP stack, derived from Linux kernel, it does not support socket API nor does it have a mechanism to address the selective state sharing problem described in Section II.

ns-3 networking stack is derived from Network Simulation Cradle (NSC) project [18]. NSC [19] uses perl scripts to parse C source files to make transformations similar to what was described in Section II.D. This approach does not scale, since pattern match and replace scripts have to be written for each application that needs to be ported.
Moreover, solution is also not feasible for languages like C++ which are more complicated. Our solution works at IR level and uses the LLVM compiler infrastructure. It works for every language that LLVM supports include C, C++ and Fortran. No special per application customization has to be done to our IR rewriter.

Time jails [20] and [21] use virtual machines (VM) like Xen and BSD jails to run real-world network applications and overcome the state sharing problem. Virtualization technologies like Xen are heavy-weight and consume significant resources. We can run only tens of VMs instead of hundreds possible by virtualizing only the network stack.

Zheng and Nicol [16] use container based OS level virtualization. Containers are lighter on resource consumption and it is possible to run a couple of hundreds of containers on same hardware which can run only ten or so VMs. The downside is that all containers use the same underlying OS network stack; hence it is not possible to configure the network stack of individual hosts differently. This would preclude certain kinds of experiment setups where different end hosts use different congestion control or active queue management algorithms. As shown in the results section V, our solution has lower memory footprint than even container based VMs.

All VM based schemes e.g. [21], [16] need modifications to the underlying OS kernel or hypervisor to implement time dilation, while our work operates purely in user land.

DieCast [21] uses uniform time dilation on end hosts to simulate faster machines than physically available. ONE uses time dilation, to emulate higher number of same capacity nodes. It also warps past idle states in the simulation without effecting temporal fidelity.

B. Testbeds and Emulators

PlanetLab [3], GENI [22], Emulab [5] and ModelNet [23] can be used as both testbed and emulator, depending on user configuration. Emulators run in real time, limiting the scale of experiments to whatever the underlying hardware provides. The effect of competing flows from unrelated network traffic is hard to isolate. Dummynet [7], NistNet [8] are single link emulators, while Emulab (which uses Dummynet) and ModelNet allow for traffic shaping along multiple hops of the network path. Nussbaum and Richard [12] contains a comparison of three widely used link emulators, Dummynet, NISTNet and Linux traffic control (tc) sub-system. They show the impact of timer frequency on emulated links.

C. Parallel Network Simulators and Emulator

A number of parallel network simulators, have been developed as research projects. Parallel and Distributed NS (PDNS) [24] uses multiple instances of a serial simulator (ns-2) to simulate disjoint parts of a network. It uses federation based interface to combine results from these disjoint simulations. ns-3’s parallel simulation is also based on same approach.

Compass [25] is another approach that combines heterogeneous simulators like ns-2 and GloMoSim [26] to take advantage of their differing strengths. It combines the transport models of ns-2 with wireless models of GloMoSim. The research goal of many of these efforts is to support reuse and composability of simulation models. All of them rely on lookahead based conservative synchronization algorithms to provide a notion of GVT.

VII. CONCLUSIONS

In this work, we presented ONE, that combines the controllability and scalability of simulation with the native code execution properties of simulators and testbeds. By combining time-dilation with virtualized network stack, ONE enables highly scalable emulations, with temporal fidelity. Our preliminary performance evaluation of ONE on an 8 core system shows that ONE is highly efficient and can run over 450 virtual hosts connected over point-to-point gigabit links, while still retaining linear behavior.

We are currently working a distributed version of time dilation and further validation of the simulator.

REFERENCES

discrete event simulations,” *Journal of the ACM (JACM)*, vol. 40,


implementation for the simulation event set problem,”

OpenVZ-Based Network Emulations,” in *2011 IEEE Workshop on
Principles of Advanced and Distributed Simulation (PADS)*, 2011,
pp. 1–10.

goals,” in *Proceeding from the 2006 workshop on ns-2: the IP

stacks,” in *Proceedings of the 37th conference on Winter


Hybrid Approach to Scalable Network Emulation,” in *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed

G. M. Voelker, “To infinity and beyond: time warped network
emulation,” in *Proceedings of the twentieth ACM symposium on
Operating systems principles*, 2005, pp. 1–2.

platform,” in *Proceedings of the 2006 ACM/IEEE symposium on
Architecture for networking and communications systems*, 2006,
pp. 1–10.

[23] V. Amin, Y. Ken, W. Kevin, M. Priya, K. Dejan, C. Jeff, and B.
David, “Scalability and accuracy in a large-scale network

www.cc.gatech.edu/computing/compass/pdns/.

and D. Xu, “A federated approach to distributed network

parallel simulation of large-scale wireless networks,” in *Proceedings of the twelfth workshop on Parallel and distributed