To Infinity and Beyond: Time-Warped Network Emulation

Diwaker Gupta, Kenneth Yocum, Marvin McNett,
Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker
University of California, San Diego
{dgupta,kyocum,mmcnett,snoeren,vahdat,voelker}@cs.ucsd.edu

Abstract

The goal of this work is to subject unmodified applications running on commodity operating systems and stock hardware to network speeds orders of magnitude faster than available at any given point in time. This paper describes our approach to *time dilation*, a technique to uniformly and accurately slow the passage of time from the perspective of an operating system by a specified factor. As a side effect, physical devices—including the network—appear relatively faster to both applications and operating systems. Both qualitative and statistical evaluations indicate our prototype implementation is accurate across several orders of magnitude. We demonstrate time dilation's utility by conducting high-bandwidth head-to-head TCP stack comparisons and application evaluation.

1 Introduction

This work explores the viability and benefits of *time dilation*—providing the illusion to an operating system and its applications that time is passing at a rate different from physical time. For example, we may wish to convince a system that, for every 10 seconds of wall clock time, only one second of time passes in the operating system's dilated time frame. Time dilation does not, however, change the arrival rate of physical events such as those from I/O devices like a network interface or disk controller. Hence, from the operating system's perspective, physical resources appear 10 times faster: in particular, data arriving from a network interface at a physical rate of 1 Gbps appears to the OS to be arriving at 10 Gbps.

We refer to the ratio between the rate at which time passes in the physical world to the operating system's perception of time as the *time dilation factor*, or TDF; a TDF greater than one indicates the external world appears faster than it really is. Figure 1 illustrates the difference between an undilated (TDF of 1) operating system on the left and a dilated OS with TDF of 2 on the right. The same period of physical time passes for both machines. Each OS receives external events such as timer (on top) and device (on bottom) interrupts. Timer interrupts update the operating systems' notion of time; in this example, time dilation halves the frequency of delivered timer interrupts.

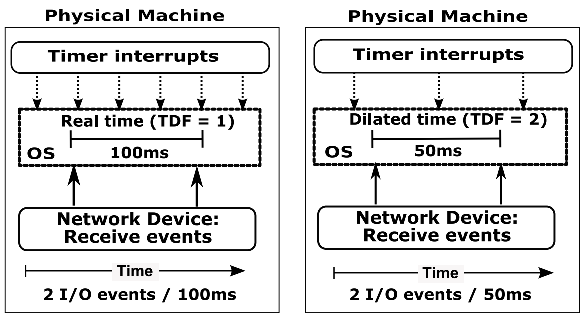


Figure 1: This figure compares a system operating in real time (left) and a system running with a TDF of 2 (right). Note that time dilation does not affect the timing of external events, such as network packet arrival.

Critically, physical devices such as the network continue to deliver events at the same rate to both OSes. The dilated OS, therefore, perceives twice the network I/O rate because it experiences only half the delay between I/O events. In particular, the "undilated" OS observes a delay of 100 ms between packet arrivals, while the dilated OS observes only 50 ms between packet arrivals. From the dilated frame of reference, time dilation scales the observed I/O rate by the TDF (in this case two).

Interestingly, time dilation scales the perceived available *processing* power as well. A system will experience TDF times as many cycles per perceived second from the processor. Such *CPU scaling* is particularly relevant to CPU-bound network processing because the number of cycles available to each arriving byte remains constant. For instance, a machine with TDF of 10 sees a 10 times faster network, but would also experience a tenfold increase in CPU cycles per second.

By employing large TDF values, time dilation enables external stimuli to appear to take place at higher rates than would be physically possible, presenting a number of interesting applications. Consider the following scenarios:

- **Emerging I/O technologies.** Imagine a complex cluster-based service interconnected by 100-Mbps and 1-Gbps Ethernet switches. The system developers suspect overall service performance is limited by network performance. However, upgrading to 10-GigE switches and interfaces involves substantial expense and overhead. The developers desire a low-cost mechanism for determining the potential benefits of higher performance network interconnects before committing to the upgrade.

- **Scalable network emulation.** Today large ISPs cannot evaluate the effects of modifications to their topology or traffic patterns outside of complex and high-level simulations. While they would like to evaluate internal network behavior driven by realistic traffic traces, this often requires accurate emulation of terabits per second of bisection bandwidth.

- **High bandwidth-delay networking.** We have recently seen the emergence of computational grids [4, 12] inter-connected by high-speed and high-latency wide-area interconnects. For instance, 10-Gbps links with 100–200 ms round-trip times are currently feasible. Unfortunately, existing transport protocols, such as TCP, deliver limited throughput to one or more flows sharing such a link. A number of research efforts have proposed novel protocols for high bandwidth-delay product settings [11, 14, 16–18, 28, 30]. However, evaluation of the benefits of such efforts is typically relegated to simulation or to those with access to expensive wide-area links.

This work develops techniques to perform accurate time dilation for unmodified applications running on commodity operating systems and stock hardware with the goal of supporting the above scenarios. To facilitate dilating time perceived by a host, we utilize virtual machine (VM) technology. Virtual machines traditionally have been used for their isolation capabilities: applications and operating systems running inside a VM can only access physical hardware through the virtual machine monitor. This interposition provides the additional potential to independently dilate time for each hosted virtual machine. In this paper, we are particularly interested in time dilation with respect to network devices; we leave faithful scaling of other subsystems to future work.

This paper makes the following contributions:

- **Time-dilated virtual machines.** We show how to use virtual machines to completely encapsulate a host running a commodity operating system in an arbitrarily dilated time frame. We allow processing power and I/O performance to be scaled independently (e.g., to hold processing power constant while scaling I/O performance by a factor of 10, or vice versa).

- **Accurate network dilation.** We perform a detailed comparison of TCP's complex end-to-end protocol behavior—in isolation, under loss, and with competing flows—under dilated and real time frames. We find that both the micro and macro behavior of the system are indistinguishable under dilation. To demonstrate our ability to predict the performance of future hardware scenarios, we show that the time-dilated performance of an appropriately dilated six-year old machine with 100-Mbps Ethernet is indistinguishable from a modern machine with Gigabit Ethernet.

- **End-to-end experimentation.** Finally we demonstrate the utility of time dilation by experimenting with a content delivery overlay service. In particular, we explore the impact of high-bandwidth network topologies on the performance of BitTorrent [9], emulating multi-gigabit bisection bandwidths using a traffic shaper whose physical capacity is limited to 1Gbps.

The remainder of the paper is organized as follows: We present our prototype implementation in Section 2. We evaluate the accuracy of time dilation with comprehensive micro-benchmarks in Section 3 and present application results in Section 4. Section 5 presents related work before concluding in Section 6.

2 Design and implementation

Before describing the details of our implementation, we first define some key terminology. A *Virtual Machine (VM)* or *domain* is a software layer that exports the interface of a target physical machine. Multiple VM's may be multiplexed on a single physical machine. A *Guest OS* is the Operating System that runs within a VM. Finally, a *Virtual Machine Monitor (VMM)* or *hypervisor* is the hardware/software layer responsible for multiplexing several VMs on the same physical machine.

Critical to time dilation is a VMM's ability to modify the perception of time within a guest OS. Fortunately, VMMs must already perform this functionality, for example, because a guest OS may develop a backlog of "lost ticks" if it is not scheduled on the physical processor when it is due to receive a timer interrupt. VMMs typically periodically synchronize the guest OS time with the physical machine's clock. One challenge is that operating systems often use multiple time sources for better precision. For example, Linux uses up to five different time sources [19]. Exposing so many different mechanisms for time keeping in virtual machines becomes challenging (see [27] for a discussion).

To be useful, time dilation must be pervasive and transparent. Pervasiveness implies that the system is completely isolated from the passage of physical time. Similarly, transparency implies that network protocols and applications require no modification to be used in a dilated time frame. To address these requirements, we implemented a time dilation prototype using the Xen VMM [7] (Our implementation is based on Xen 2.0.7). We chose Xen for the following reasons: (1) it is easier to modify behavior of timer interrupts in software than in hardware; (2) we can observe time dilation in isolated, controlled environments; (3) Xen allows us to provide each virtual machine with an independent time frame; (4) Xen source code is publicly available; and iv) the VMM CPU scheduler provides a facility for scaling CPU. Though our implementation is Xen-specific, we believe the concepts can apply to other virtual machine environments.

Alternative implementation targets for time dilation include directly modifying the operating system, simulation packages, and emulation environments. As discussed below, our modifications to Xen are compact and portable, giving us confidence that our techniques will be applicable to any operating system that Xen supports. In some sense, time dilation is free in many simulation packages: extrapolating to future scenarios is as simple as setting appropriate bandwidth values on particular links. However, we explicitly target running unmodified applications and operating systems for necessary realism. Finally, while network emulation does allow experimentation with a range of network conditions, it is necessarily limited by the performance of currently available hardware. For this reason, time dilation is a valuable complement to network emulation, allowing an experimenter to easily extrapolate evaluations to future, faster environments.

We now give a brief overview of time keeping in Xen, describe our modifications to it, and discus the applicability of time dilations to other virtualization platforms.

2.1 Time flow in Xen

The Xen VMM exposes two notions of time to VMs. *Real time* is the number of nanoseconds since boot. *Wall clock time* is the traditional Unix time-since-epoch (midnight, January 1, 1970 UTC). Xen also delivers periodic timer interrupts to the VM to support the time keeping mechanisms within the guest OS.

While Xen allows the guest OS to maintain and update its own notion of time via an external time source (such as NTP), the guest OS often relies solely on Xen to maintain accurate time. Real and wall clock time pass between the Xen hypervisor and the guest operating system via a shared data structure. There is one data structure per VM written by the VMM and read by the guest OS.

The guest operating system updates its local time values on demand using the shared data structure — for instance, when servicing timer interrupts or calling `getttimeofday`. However, the VMM updates the shared data structure only at certain discrete events, and thus it may not always contain the current value. In particular, the VMM updates the shared data structure when it delivers a timer interrupt to the VM or schedules the VM to run on an available CPU.

Xen uses *paravirtualization* to achieve scalable performance with virtual machines without sacrificing functionality or isolation. With paravirtualization, Xen does not provide a perfect virtualization layer. Instead, it exposes some features of the underlying physical hardware to gain significant performance benefits. For instance, on the x86 architecture, Xen allows guest OSes (for our tests, we use XenoLinux as our guest OS) to read the Time-Stamp Count (TSC) register directly from hardware (via the `RDTSC` instruction).

The TSC register stores the number of clock cycles since boot and is incremented on every CPU cycle. The Guest OS reads the TSC to maintain accurate time between timer interrupts. By contrast, kernel variables such as Linux `jiffies` or BSD `ticks` only advance on timer interrupts. In this case, we modify the guest OS to prevent them from reading the true value of the TSC, as described in the next section.

2.2 Dilating time in Xen

We now outline our modifications to the Xen hypervisor and the XenoLinux kernel to support time dilation. We focus on slowing down the passage of time so that the external world appears faster. It is also possible to speed the passage of time from the OS's perspective, thereby slowing processes in the external world. In general, however, speeding the passage of time is less useful for our target scenarios and we do not explore it in this paper.

Our modifications to Xen are small: in all, we added/modified approximately 500 lines of C and Python code. More than 50% of our changes are to non-critical tools and utilities; the core changes to Xen and XenoLinux are less than 200 lines of code. Our modifications are less than 0.5% of the base code size of each component.

Variable	Original	Dilated
Real time	`stime_irq`	`stime_irq/tdf`
Wall clock	`wc_sec`, `wc_usec`	`wc_sec/tdf`, `wc_usec/tdf`
Timer interrupts	`HZ/sec`	`(HZ/tdf)/sec`

Table 1: Basic Dilation Summary

Modifications to the Xen hypervisor. Our modified VMM maintains a TDF variable for each hosted VM. For our applications, we are concerned with the relative passage of time rather than the absolute value of real time; in particular, we allow—indeed, require—that the host's view of wall clock time diverge from reality. Thus the TDF divides both real and wall clock time.

We modify two aspects of the Xen hypervisor. First we extend the shared data structure to include the TDF field. Our modified Xen tools, such as `xm`, allow specifying a positive, integral value for the TDF on VM creation. When the hypervisor updates the shared data structure, it uses this TDF value to modify real and wall clock time. In this way, real time is never exposed to the guest OS through the shared data structure.

Dilation also impacts the frequency of timer interrupts delivered to the VM. The VMM controls the frequency of timer interrupts delivered to an undilated VM (timer interrupts/second); in most OS's a HZ variable, set at compile time, defines the number of timer interrupts delivered per second. For transparency, we need to maintain the invariant that HZ accurately reflects the number of timer interrupts delivered to the VM during a second of dilated time. Without adjusting timer interrupt frequency, the VMM will deliver TDF-times too many interrupts. For example, the VMM will deliver HZ interrupts in one physical time second, which will look to the dilated VMM as HZ/(second/TDF) = TDF*HZ. Instead, we reduce the number of interrupts a VM receives by a factor of TDF (as illustrated earlier in Figure 1). Table 1 summarizes the discussion so far.

Finally, Xen runs with a default HZ value of 100, and configures guests with the same value. However, $HZ = 100$ gives only a 10-ms precision on system timer events. In contrast, current 2.6 series of Linux kernels uses a HZ value of 1000 by default—the CPU overhead is not significant, but the accuracy gains are tenfold. This increase in accuracy is desirable for time dilation because it enables guests to measure time accurately even in the dilated time frame. Thus, we increase the HZ value to 1000 in both Xen and the guest OS.

Modifications to XenoLinux. One goal of our implementation was to minimize required modifications to the guest OS. Because the VMM appropriately updates the shared data structure, one primary aspect of OS timekeeping is already addressed. We further modify the guest OS to read an appropriately scaled version of the hardware Time Stamp Counter (TSC). XenoLinux now reads the TDF from the shared data structure and adjusts the TSC value in the function `get_offset_tsc`.

The Xen hypervisor also provides guest OS programmable alarm timers. Our last modification to the guest OS adjusts the programmable timer events. Because guests specify timeout values in their dilated time frames, we must scale the timeout value back to physical time. Otherwise they may set the timer for the wrong (possibly past) time.

2.3 Time dilation on other platforms

Architectures: Our implementation should work on all platforms supported by Xen. One remark regarding transparency of time dilation to user applications on the x86 platform is in order: recall that we intercept calls to read the TSC within the guest kernel. However, since the `RDTSC` instruction is not a privileged x86 instruction, guest user applications might still issue assembly instructions to read the hardware TSC register. It is possible to work around this by watching the instruction stream emanating from a VM and trapping to the VMM on a `RDTSC` instruction, and then returning the appropriate value to the VM. However, this approach would go against Xen's paravirtualization philosophy.

Fortunately, the current generation of x86-compatible hardware (such as the AMD Pacifica [6] and Intel VT [13]) provides native virtualization support, making it possible to make time dilation completely transparent to applications. For instance, both VT and Pacifica have hardware support for trapping the `RDTSC` instruction.

VMMs: The only fundamental requirement from a VMM for supporting time dilation is that it have mechanisms to update/modify time perceived by a VM. As mentioned earlier, due to the difficulties in maintaining time within a VM, all VMMs already have similar mechanisms so that they can periodically bring the guest OS time in sync with real time. For instance, VMWare has explicit support for keeping VMs in a "fictitious time frame" that is at a constant offset from real time [27]. Thus, it should be straightforward to implement time dilation for other VMMs.

Operating systems: Our current implementation provides dilation support for XenoLinux. Our experience so far and a preliminary inspection of the code for other guest OSes indicate that all of the guest OSes that Xen supports can be easily modified to support time dilation. It is important to note that modifying the guest OSes is not a fundamental requirement. Using binary rewriting, it would be possible to use unmodified guest OS executables. We expect that with better hardware and operating system support for virtualization, unmodified guests would be able to run under dilation.

2.4 Limitations

This section discusses some of the limitations of time dilation. One obvious limitation is time itself: a 10-second experiment would run for a 100 seconds for a dilation factor of 10. Real-life experiments running for hours are not uncommon, so the time required to run experiments at high TDFs is substantial. Below we discuss other, more subtle limitations.

2.4.1 Other devices and nonlinear scaling

Time dilation uniformly scales the perceived performance of all system devices, including network bandwidth, perceived CPU processing power, and disk and memory I/O. Unfortunately, scaling all aspects of the physical world is unlikely to be useful: a researcher may wish to focus on scaling certain features (e.g., network bandwidth) while leaving others constant. Consequently, certain aspects of the physical world may need to be rescaled accordingly to achieve the desired effect.

Consider TCP, a protocol that depends on observed round-trip times to correctly compute retransmission timeouts. These timing measurements must be made in the dilated time frame. Because time dilation uniformly scales the passage of time, it not only increases perceived bandwidth, it also decreases perceptions of round-trip time. Thus, a network with 10-ms physical RTT would appear to have 1-ms RTT to dilated TCP. Because TCP performance is sensitive to RTT, such a configuration is likely undesirable. To address this effect, we independently scale bandwidth and RTT by using network emulation [23, 26] to deliver appropriate bandwidth and latency values. In this example, we increase link delay by a factor of 10 to emulate the jump in bandwidth-delay product one expects from the bandwidth increase.

In this paper, we show how to apply time dilation to extrapolate to future network environments, for instance with a factor of 10 or 100 additional bandwidth while accounting for variations in CPU power using the VMM scheduler. However, we do not currently account for the effects of increased I/O rates from memory and disk.

Appropriately scaling disk performance is a research challenge in its own right. Disk performance depends on such factors as head and track-switch time, SCSI-bus overhead, controller overhead, and rotational latency. A simple starting point would be to vary disk performance assuming a linear scaling model, but this could potentially violate physical properties inherent in disk drive mechanics. Just as we introduced appropriate network delays to account for non-linear scaling of the network, accurate disk scaling would require modifying the virtual machine monitor to integrate a disk simulator modified to understand the dilated time frame. A well-validated disk simulator, such as DiskSim [8], could be used for this purpose. However, we leave dilating time for such devices to future work.

Finally, hardware and software architectures may evolve in ways that time dilation cannot support. For instance, consider a future host architecture with TCP offload [20], where TCP processing largely takes place on the network interface rather than in the protocol stack running in the operating system. Our current implementation does not dilate time for firmware on network interfaces, and may not extend to other similar architectures.

2.4.2 Timer interrupts

The guest reads time values from Xen through a shared data structure. Xen updates this structure every time it delivers a timer interrupt to the guest. This happens on the following events: (1) when a domain is scheduled; (2) when the *currently executing* domain receives a periodic timer interrupt; and (3) when a guest-programmable timer expires.

We argued earlier that, for successful dilation, the number of timer interrupts delivered to a guest should be scaled down by the TDF. Of these three cases, we can only control the second and the third. Our implementation does not change the scheduling pattern of the virtual machines for two reasons. First, we do not change the schedule pattern because scheduling parameters are usually global and system wide. Scaling these parameters on a per-domain basis would likely break the semantics of many scheduling schemes. Second, we would like to retain scheduling as an independent variable in our system, and therefore not constrain it by the choice of TDF. One might want to use round robin or borrowed virtual time [10] as the scheduling scheme, independent of the TDF. In practice, however, we find that not controlling timer scheduling does not impact our accuracy for all of the schedulers that currently ship with Xen.

2.4.3 Uniformity: Outside the dilation envelope

Time dilation cannot affect notions of time outside the dilation envelope. This is an important limitation; we should account for all packet processing or delays external to the VM. The intuition is that all stages of packet processing should be uniformly dilated. In particular, we scale the time a packet spends inside the VM (since it measures time in the dilated frame) and the time a packet spends over the network (by scaling up the time on the wire by TDF). However, we do not scale the time a packet spends inside the Xen hypervisor and *Domain-0* (the privileged, management domain that hosts the actual device drivers), or the time it takes to process the packet at the other end of the connection.

These unscaled components may affect the OS's interpretation of round trip time. Consider the time interval between a packet and its ACK across a link of latency R scaled by S, and let δ denote the portion of this time that is unscaled. In a perfect world where everything is dilated uniformly, a dilated host would measure the interval to be simply $T_{perfect} = R + \delta$. A regular, undilated host measures the interval as $T_{normal} = S \times R + \delta$; a dilated host in our implementation would observe the same scaled by S, so $T_{dilated} = T_{normal}/S = (S \times R + \delta)/S$.

We are interested in the error relative to perfect dilation:

$$\begin{aligned}\epsilon &= \frac{(T_{perfect} - T_{dilated})}{T_{perfect}} \\ &= \left(1 - \frac{1}{S}\right)\left(\frac{\delta}{R+\delta}\right)\end{aligned}$$

Note that ϵ approaches $\delta/(R + \delta)$ when S is large. In the common case this is of little consequence. For the regime of network configurations we are most interested in (high bandwidth-delay product networks), the value of R is typically orders of magnitude higher than the value of δ. As our results in Section 3 show, dilation remains accurate over a wide range of round trip times, bandwidths, and time-dilation factors that we consider.

3 Micro-benchmarks

We establish the accuracy of time dilation through a variety of micro-benchmarks. We begin by evaluating the accuracy of time dilation by comparing predictive results using dilated older hardware with actual results using undilated recent hardware. We then compare the behavior of a single TCP flow subject to various network conditions under different time dilation factors. Finally, we evaluate time dilation in more complex settings (multiple flows, multiple machines) and address the impact of CPU scaling.

3.1 Hardware validation

We start by evaluating the predictive accuracy of time dilation using multiple generations of hardware. One of the key motivations for time dilation is as a predictive tool, such as predicting the performance and behavior of protocol and application implementations on future higher-performance network hardware. To validate time dilation's predictive accuracy, we use dilation on older hardware to predict TCP throughput as if we were using recent hardware. We then compare the predicted performance with the actual performance when using recent hardware.

Configuration	**TDF**	**Mean (Mbps)**	**St.Dev. (Mbps)**
2.6 GHz, 1-Gbps NIC (restricted to 500-Mbps)	1	9.39	1.91
1.13 GHz, 1-Gbps NIC (restricted to 250-Mbps)	2	9.57	1.76
500 MHz, 100-Mbps NIC	5	9.70	2.04
500 MHz scaled down to 50 MHz, 10-Mbps NIC	50	9.25	2.20

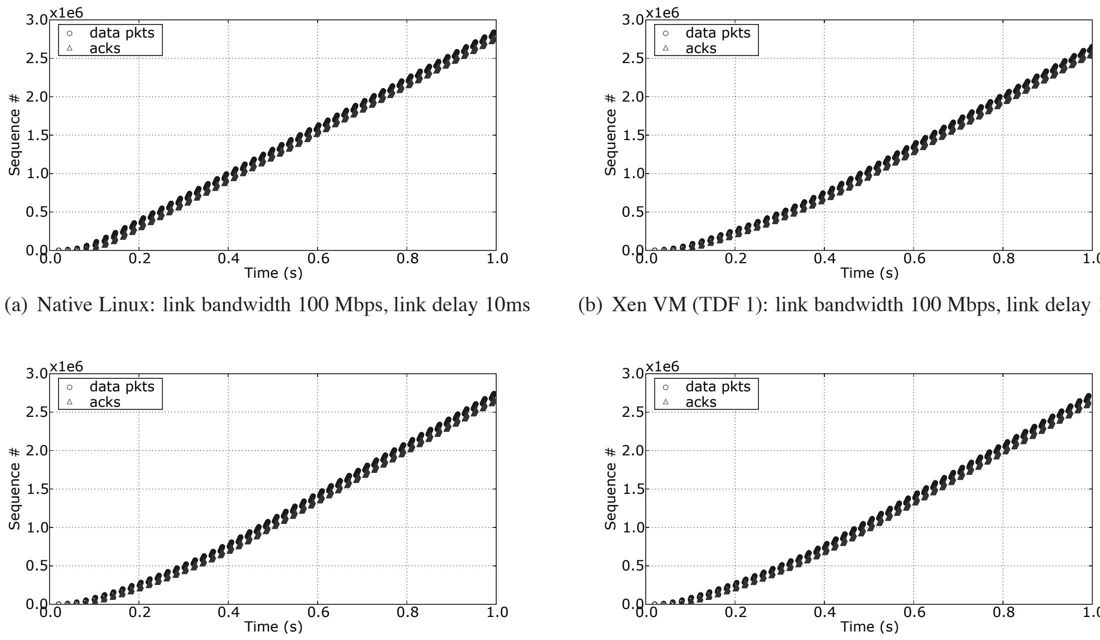
Table 2: Validating performance prediction: the mean per-flow throughput and standard deviations of 50 TCP flows for different hardware configurations.

We use time dilation on four hardware configurations, listed in Table 2, such that each configuration resembles a 2.5-GHz processor with a 500-Mbps NIC, under the coarse assumption that CPU performance roughly scales with processor frequency. The base hardware configurations are 500-MHz and 1.13-GHz Pentium III machines with 10/100-Mbps and 1-Gbps network interfaces, respectively, and a 2.6-GHz Pentium IV machine with a 1-Gbps network interface. In cases where the exact base hardware was not available (a 250-Mbps NIC or a 50-MHz CPU, for instance), we scaled them appropriately using either network emulation (Dummynet [23]) or VMM scheduling (Section 3.4).

For each hardware configuration, we measured the TCP throughput of 50 flows communicating with another machine with an identical configuration. Using Dummynet, we configured the network between the hosts to have an effective RTT of 80 ms. We then calculated the mean per-flow throughput and standard deviation across all flows. Both the mean and deviation of per-flow throughput are consistent across the hardware configurations, which span over an order of magnitude of difference in hardware performance. For example, time dilation using an effective 50 MHz CPU with a 10-Mbps NIC dilated with a TDF of 50 is able to accurately predict TCP throughput of a 2.6-GHz CPU with a 1-Gbps NIC. As a result, we conclude that time dilation is an effective tool for making reasonable predictions of high-level performance on future hardware trends.

3.2 Single flow packet-level behavior

Next we illustrate that time dilation preserves the perceived packet-level timing of TCP. We use two end hosts directly connected through a Dell Powerconnect 5224 gigabit switch. Both systems are Dell PowerEdge 1750 servers with dual Intel 2.8-GHz Xeon processors, 1 GB of physical memory, and Broadcom NetXtreme



(a) Native Linux: link bandwidth 100 Mbps, link delay 10ms (b) Xen VM (TDF 1): link bandwidth 100 Mbps, link delay 10 ms

(c) Xen VM (TDF 10): link bandwidth 10 Mbps, link delay 100 ms (d) Xen VM (TDF 100): link bandwidth 1 Mbps, link delay 1000 ms

Figure 2: Packet timings for the first second of a TCP connection with no losses for native Linux and three time dilation configurations. In all cases, we configure link bandwidth and delay such that the bandwidth-delay product is constant.

BCM5704 integrated gigabit Ethernet NICs. The end hosts run Xen 2.0.7, modified to support time dilation. We use Linux 2.6.11 as the Xen guest operating system, and all experiments run inside of the Linux guest VMs. All protocols in our experiments use their default parameters unless otherwise specified. We use two identical machines running Linux 2.6.10 and Fedora Core 2 for our "unmodified Linux" results.

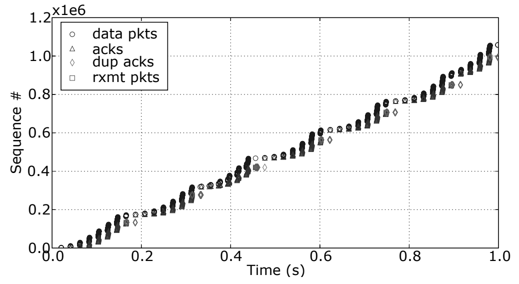
We control network characteristics such as bandwidth, delay, and loss between the two hosts using Dummynet. In addition to its random loss functionality, we extended Dummynet to support deterministic losses to produce repeatable and comparable loss behavior. Unless otherwise noted, all endpoints run with identical parameters (buffer sizes, TDFs, etc.).

In this experiment, we first transfer data on TCP connections between two unmodified Linux hosts and use `tcpdump` [5] on the sending host to record packet behavior. We measure TCP behavior under both lossless and deterministic lossy conditions.

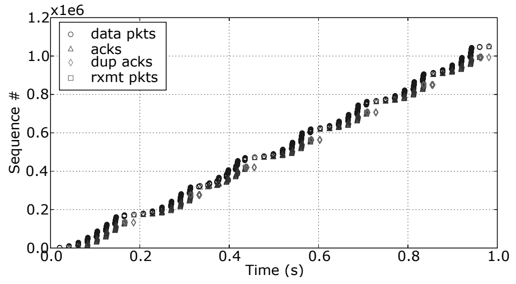
We then repeat the experiment with the sending host running with TDFs of $\{1, 10, 100\}$, spanning two orders of magnitude. When dilating time, we configure the underlying network such that a time-dilated host perceives the same network conditions as the original TCP experiment on unmodified hosts. For example, for the experiment with unmodified hosts, we set the bandwidth between the hosts to 100 Mbps and the delay to 10 ms. To preserve the same network conditions perceived by a host dilated by a factor of 10, we reduce the bandwidth to 10 Mbps and increase the delay to 100 ms using Dummynet. Thus, if we are successful, a time dilated host will see the same packet timing behavior as the unmodified case. We include results with TDF of 1 to compare TCP behavior in an unmodified host with the behavior of our system modified to support time dilation.

We show sets of four time sequence graphs in Figures 2 and 3. Each graph shows the packet-level timing behavior of TCP on four system configurations: unmodified Linux, and Linux running in Xen with our implementation of time dilation operating under TDFs of 1, 10, and 100. The first set of graphs shows the first second of a trace of TCP without loss. Each graph shows the data and ACK packet sequences over time, and illustrates TCP slow-start and transition to steady-state behavior. Qualitatively, the TCP flows across configurations have nearly identical behavior.

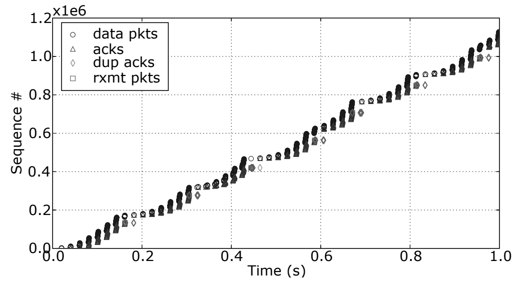
Comparing Figures 2(a) and 2(b), we see that a dilated host has the same packet-level timing behavior as an unmodified host. More importantly, we see that time dilation accurately preserves packet-level timings perceived by the dilated host. Even though the configuration with a TDF of 100 has network conditions two orders of magnitude different from the base configuration, time dilation successfully preserves packet-level timings.



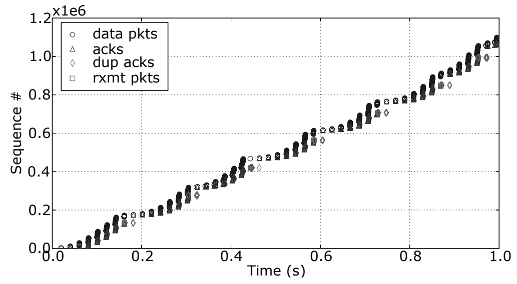
(a) Native Linux: link bandwidth 100 Mbps, link delay 10 ms



(b) Xen VM (TDF 1): link bandwidth 100 Mbps, link delay 10 ms



(c) Xen VM (TDF 10): link bandwidth 10 Mbps, link delay 100 ms



(d) Xen VM (TDF 100): link bandwidth 1 Mbps, link delay 1000 ms

Figure 3: Packet timings for the first second of a TCP connection with 1% deterministic losses.

Time dilation also accurately preserves packet-level timings under lossy conditions. The second set of time sequence graphs in Figure 3 shows the first second of traces of TCP experiencing 1% loss. As with the lossless experiments, the TCP flows across configurations have nearly identical behavior even with orders of magnitude differences in network conditions.

We further evaluated the performance of a single TCP flow under a wide range of time dilation factors, network bandwidths, delays and loss rates with similar results. For brevity, we omit those results.

Figures 2 and 3 illustrate the accuracy of time dilation qualitatively. For a more quantitative analysis, we compared the distribution of the inter-arrival packet reception and transmission times for the dilated and undilated flows. Figure 4 plots the cumulative distribution function for inter-packet transmission times for a single TCP flow across 10 runs under both lossy and lossless conditions. Visually, the distributions track closely. Table 3 presents a statistical summary for these distributions, the mean and two indices of dispersion — the coefficient of variance (CoV) and the inter quartile range (IQR) [15]. An index of dispersion indicates the variability in the given data set. Both CoV and IQR are unit-less, i.e., they take the unit of measurement out of variability consideration. Therefore, the absolute values of these metrics is not of concern to us, but that they match under dilation is. Given the inherent variability in TCP, we find this similarity satisfactory. The results for inter-packet reception times are similar.

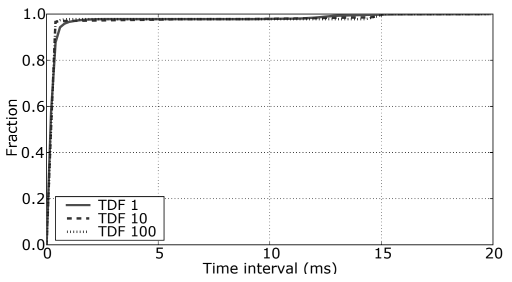
Metric	**No loss**			1% loss		
	TDF 1	TDF 10	TDF 100	TDF 1	TDF 10	TDF 100
Mean (ms)	0.458	0.451	0.448	0.912	1.002	0.896
CoV	0.242	0.218	0.206	0.298	0.304	0.301
IQR	0.294	0.248	0.239	0.202	0.238	0.238

Table 3: Statistical summary of inter-packet transmission times.

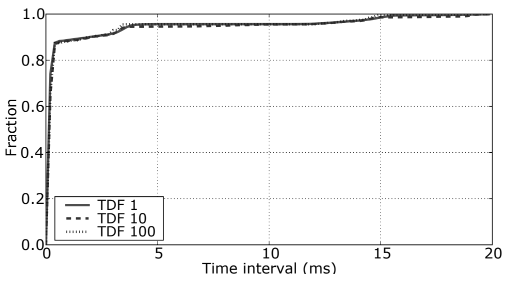
3.3 Dilation with multiple flows

To demonstrate that dilation preserves TCP behavior under a variety of conditions, even for short flows, we performed another set of experiments under heterogeneous conditions. In these experiments, 60 flows shared a bottleneck link. We divided the flows into three groups of 20 flows, with each group subject to an RTT of 20 ms, 40 ms, or 60 ms. We also varied the bandwidth of the bottleneck link from 10 Mbps to 600 Mbps. We conducted the experiments for a range of flow lengths from 5 seconds to 60 seconds and verified that the results were consistent independent of flow duration.

We present data for one set of experiments where each flow lasts for 10 seconds. Figure 5 plots the mean and standard deviation across the flows within each group for TDFs of 1 (regular TCP) and 10. To visually differentiate results in each graph for different TDFs, we slightly offset their error bars on the graph although in practice they experienced the same network bandwidth conditions. For all three groups, the results from dilation



(a) Distribution under no loss



(b) Distribution under 1% loss

Figure 4: Comparison of inter-packet transmission times for a single TCP flow across 10 runs.

agree well with the undilated results: the throughputs for TDF of 1 match those for TDF of 10 within measured variability. Note that these results also reflect TCP's known throughput bias towards flows with smaller RTTs.

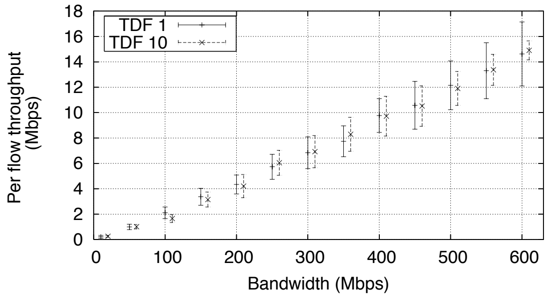
In our experiments thus far, all flows originated at a single VM and were destined to a single VM. However, when running multiple VMs (as might be the case to support, for instance, scalable network emulation experiments [26, 29]) one has to consider the impact of VMM scheduling overhead on performance. To explore the issue of VMM scheduling, we investigate the impact of spreading flows across multiple dilated virtual machines running on the same physical host. In particular, we verify that simultaneously scheduling multiple dilated VMs does not negatively impact the accuracy of dilation.

In this experiment, for a given network bandwidth we create 50 connections between two hosts with a lifetime of 1000 RTTs and measure the resulting throughput of each connection. We configure the network with an 80 ms RTT, and vary the perceived network bandwidth from 0–4 Gbps using 1-Gbps switched Ethernet. Undilated TCP has a maximum network bandwidth of 1 Gbps, but time dilation enables us to explore performance beyond the raw hardware limits (we revisit this point in Section 4.1). We repeat this experiment with the 50 flows split across 2, 5 and 10 virtual machines running on one physical machine.

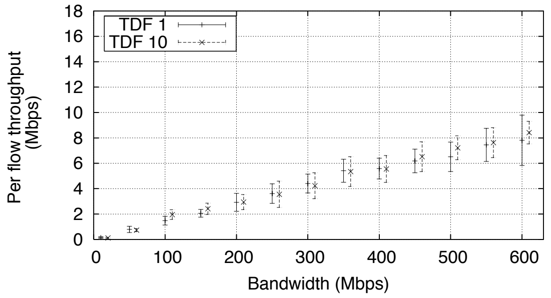
Our results indicate that VMM scheduling does not significantly impact the accuracy of dilation. Figure 6 plots the mean throughput of the flows for each of the four configurations of flows divided among virtual machines. Error bars mark the standard deviation. Once again, the mean flow throughput for the various configurations are similar.

3.4 CPU scaling

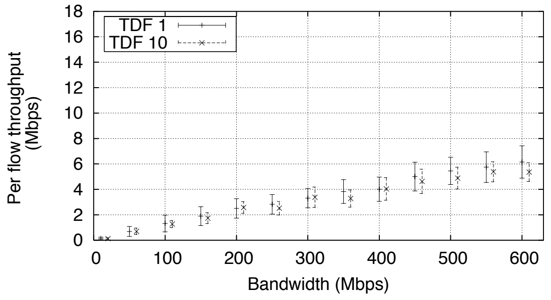
Time dilation changes the perceived VM cycle budget; a dilated virtual machine sees TDF times as many CPU cycles per second. Utilizing VMM CPU schedulers, how-



(a) 20 flows subject to 20-ms RTT



(b) 20 flows subject to 40-ms RTT



(c) 20 flows subject to 60-ms RTT

Figure 5: Per-flow throughput for 60 flows sharing a bottleneck link. Each flow lasts 10 seconds. The mean and deviation are taken across the flows within each group. To visually differentiate results in each graph for different TDFs, we slightly offset their error bars on the graph.

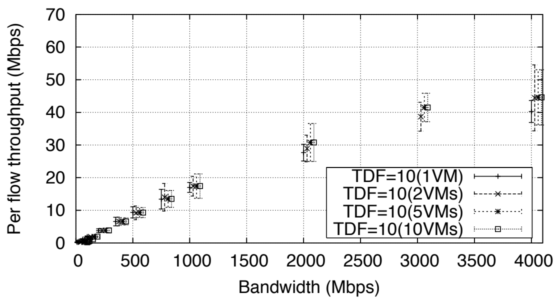


Figure 6: Mean throughput of 50 TCP flows between two hosts on a network with an 80ms RTT as a function of network bandwidth. The 50 flows are partitioned among 1–10 virtual machines.

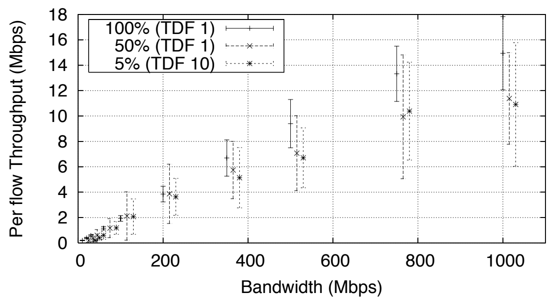


Figure 7: Per-flow throughput of 50 TCP flows between a CPU-scaled sender and unconstrained receiver. CPU utilization at the sender is restricted to the indicated percentages.

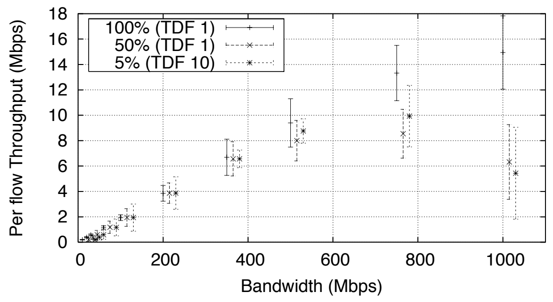


Figure 8: Per-flow throughput of 50 TCP flows between an unconstrained sender and a CPU-scaled receiver. CPU utilization at the receiver is restricted to the indicated percentages.

ever, we can scale available processing power independently from the network. This flexibility allows us to evaluate the impact of future network hardware on current processor technology. In a simple model, a VM with TDF of 10 running with 10% of the CPU has the same per-packet cycle budget as an undilated VM running with 100% of the CPU. We validate this hypothesis by running an experiment similar to that described for Figure 6. This time, however, we adjust the VMM's CPU scheduling algorithm to restrict the amount of CPU allocated to each VM. We use the Borrowed Virtual Time [10] scheduler in Xen to assign appropriate weights to each domain, and a CPU intensive job in a separate domain to consume surplus CPU.

First, we find an undilated scenario that is CPU-limited by increasing link capacity. Because the undilated processor has enough power to run the network at line speed, we reduce its CPU capacity by 50%. We compare this to a VM dilated by TDF of 10 whose CPU has been scaled to 5%. The experimental setup is identical to that in Figure 6: 50 flows, 80ms RTT. For clarity, we first throttled the sender alone, leaving the CPU unconstrained at the receiver; we then repeat the experiment with the receiver alone throttled. Figures 7 and 8 show the results. We plot the per-flow throughput, and error bars mark the standard deviation.

If we successfully scale the CPU, flows across a dilated link of the same throughput will encounter identical CPU limitations. Both figures confirm the effectiveness of CPU scaling, as the 50% and 5% lines match closely. The unscaled line (100%) illustrates the performance in a CPU-rich environment. Moreover our system accurately predicts that receiver CPU utilizations are higher than the sender's, confirming that it is possible to dilate CPU and network independently by leveraging the VMMs CPU scheduling algorithm.

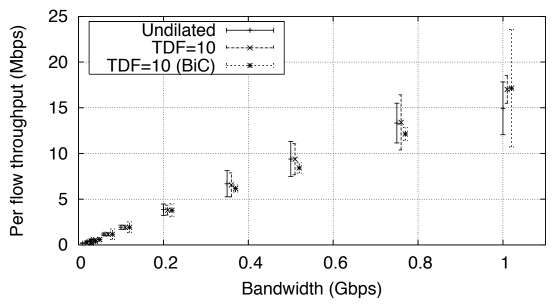
4 Applications of dilation

Having performed micro-benchmarks to validate the accuracy of time dilation, we now demonstrate the utility of time dilation for two scenarios: network protocol evaluation and high-bandwidth applications.

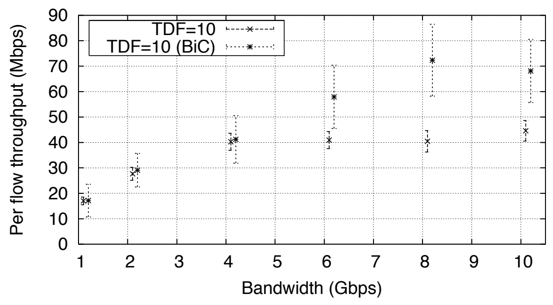
4.1 Protocol evaluation

A key application of time dilation is for evaluating the behavior and performance of protocols and their implementations on future networks. As an initial demonstration of our system's utility in this space, we show how time dilation can support evaluating optimizations to TCP for high bandwidth-delay network environments, in particular using the publicly available BiC [30] extension to the Linux TCP stack. BiC uses binary search to increase the congestion window size logarithmically — the rate of increase is higher when the current transmission rate is much less than the target rate, and slows down as it gets closer to the target rate.

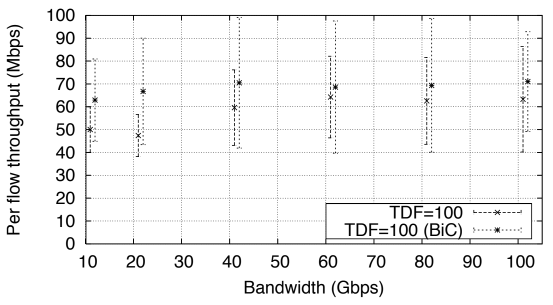
For the network configuration, we use an 80 ms RTT and vary the network bandwidth up to 100 Gbps using underlying 1-Gbps hardware. We configure the machines exactly as in Section 3.3. We perform this experiment for two different protocols: TCP, and TCP with BiC enabled



(a) Validating dilation: TCP performance under dilation matches actual, observed performance.



(b) Using dilation for protocol evaluation: comparing TCP with TCP BiC under high bandwidth.



(c) Pushing the dilation envelope: using a TDF of 100 to evaluate protocols under extremely high bandwidths.

Figure 9: Protocol Evaluation: Per-flow throughput of 50 flows for TCP and TCP BiC between two hosts on a network with an 80-ms RTT as a function of network bandwidth.

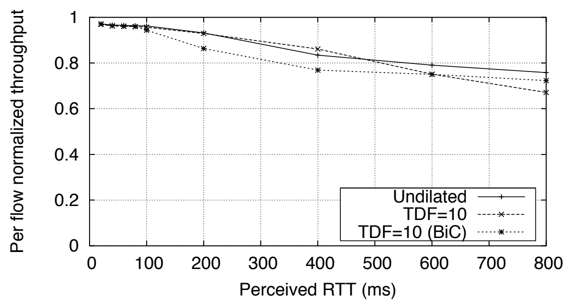


Figure 10: Protocol evaluation: Normalized average per-flow throughput of 50 flows for TCP and TCP BiC between two hosts on a network with 150 Mbps bandwidth as a function of RTT.

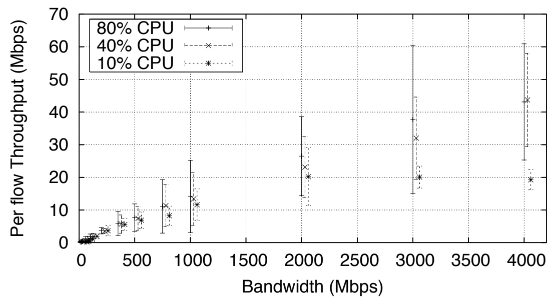


Figure 11: Per-flow throughput of 50 TCP flows across two hosts as a function of network bandwidths. CPU utilization at the sender is restricted to the indicated percentages. Experiments run with TDF of 10.

(henceforth referred to as BiC). In all of the following experiments, we adjust the Linux TCP buffers as suggested in the TCP Tuning Guide [2].

Figure 9 shows per-flow throughput of the 50 connections as a function of network bandwidth. For one execution, we plot the average throughput per flow, and the error bar marks the standard deviation across all flows. In Figure 9(a), the x-axis goes up to 1 Gbps, and represents the regime where the accuracy of time dilation can be validated against actual observations. Figures 9(b) (1 to 10 Gbps) and 9(c) (10 to 100 Gbps) show how time dilation can be used to extrapolate performance.

The graphs show three interesting results. First, time dilation enables us to experiment with protocols beyond hardware limits using implementations rather than simulations. Here we experiment with an unmodified TCP stack beyond the 1 Gbps hardware limit to 100 Gbps. Second, we can experimentally show the impact of high bandwidth-delay products on TCP implementations. Beyond 10 Gbps, per-flow TCP throughput starts to level off. Finally, we can experimentally demonstrate the benefits of new protocol implementations designed for such networks. Figure 9(b) shows that in the 1–10 Gbps regime, BiC outperforms TCP by a significant margin. However, in Figure 9(c) we see that TCP shows a steady, gradual improvement and both BiC and TCP level off beyond 10 Gbps.

TCP performance is also sensitive to RTT. To show this effect under high-bandwidth conditions, we perform another experiment with 50 connections between two machines. However, we instead fix the network bandwidth at 150 Mbps and vary the perceived RTT between the hosts. For clarity, we present an alternative visualization of the results: instead of plotting the absolute per-flow throughput values, we instead plot normalized throughput values as a fraction of maximum potential throughput. For example, with 50 connections on a 150-

Mbps bandwidth link, the maximum average per-flow throughput would be 3 Mbps. Our measured average per-flow throughput was 2.91 Mbps, resulting in a normalized per-flow throughput of 0.97. Figure 10 shows the average per-flow throughput of the three protocols as a function of RTT from 0–800 ms. We chose this configuration to match a recent study on XCP [16], a protocol targeting high bandwidth-delay conditions. The results show the well-known dependence of TCP throughput on RTT, and that the two dilated protocols behave similarly to undilated TCP.

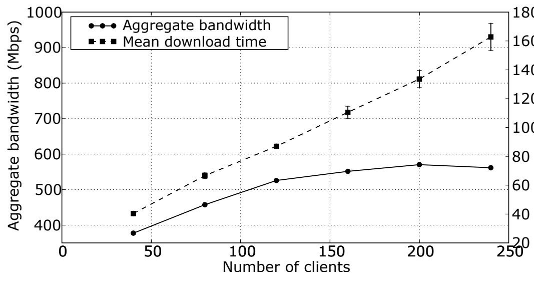
We can also use time dilation as a tool to estimate the computational power required to sustain a target bandwidth. For instance, from Figure 11, we can see that across a 4-Gbps pipe with an 80-ms RTT, 40% CPU on the sender is sufficient for TCP to reach around 50% utilization. This means that processors that are 4 times as fast as today's processors will be needed to achieve similar performance (since 40% CPU at TDF of 10 translates to 400% CPU at TDF of 1).

4.2 High-bandwidth applications

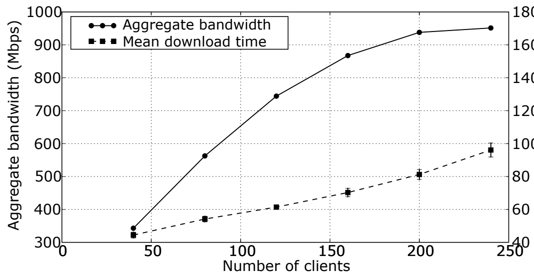
Time dilation can significantly enhance our ability to evaluate data-intensive applications with high bisection bandwidths using limited hardware resources. For instance, the recent popularity of peer-to-peer environments for content distribution and streaming requires significant aggregate bandwidth for realistic evaluations. Capturing the requirements of 10,000 hosts with an average of 1 Mbps per host would require 10 Gbps of emulation capacity and sufficient processing power for accurate study—a hardware configuration that would be prohibitively expensive to create for many research groups.

We show initial results of our ability to scale such experiments using modest hardware configurations with BitTorrent [9], a high-bandwidth peer-to-peer, content distribution protocol. Our goal was to explore the bottlenecks when running a large scale experiment using the publicly available BitTorrent implementation [1] (version 3.4.2).

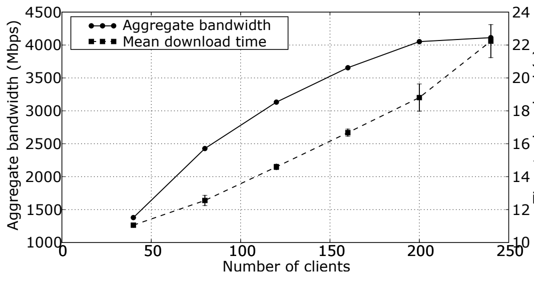
We conducted our experiments using 10 physical machines hosting VMs running BitTorrent clients interconnected through one ModelNet [26] core machine emulating an *unconstrained* network topology of 1,000 nodes. The client machines and the ModelNet core are physically connected via a gigabit switch. The ModelNet topology is unconstrained in the sense that the network emulator forwards packets as fast as possible between endpoints. We create an overlay of BitTorrent clients, all of which are downloading a 46-MB file from an initial "seeder". We vary the number of clients participating in the overlay, distributing them uniformly among the 10 VMs. As a result, the aggregate bisection bandwidth of



(a) VMs are running with TDF of 1 (no dilation). Performance degrades as clients contend for CPU resources.



(b) VMs are running with TDF of 10 and perceived network capacity is 1 Gbps. Dilation removes CPU contention, but network capacity becomes saturated with many clients.



(c) VMs are running with TDF of 10. Perceived network capacity is 10 Gbps. Increasing perceived network capacity removes network bottleneck, enabling aggregate bandwidth to scale until clients again contend for CPU.

Figure 12: Using time dilation for evaluating BitTorrent: Increasing the number of clients results in higher aggregate bandwidths, until the system reaches some bottleneck (CPU or network capacity). Time dilation can be used to push beyond these bottlenecks.

the BitTorrent overlay is limited by the emulation capacity of ModelNet, resource availability at the clients, and the capacity of the underlying hardware.

In the following experiments, we demonstrate how to use time dilation to evaluate BitTorrent performance beyond the physical resource limitations of the test-bed. As a basis, we measure a BitTorrent overlay running on the VMs with a TDF of 1 (no dilation). We scale the number of clients in the overlay from 40 to 240 (4–24 per VM). We measure the average time for downloading the file across all clients, as well as the aggregate bisection bandwidth of the overlay; we compute aggregate bandwidth as the number of clients times the average per-client bandwidth (file size/average download time). Figure 12(a) shows the mean and standard deviation for 10 runs of this experiment as a function of the number of clients. Since the VMs are not dilated, the aggregate bisection bandwidth cannot exceed the 1-Gbps limit of the physical network. From the graph, though, we see that the overlay does not reach this limit; with 200 clients or more, BitTorrent is able to sustain aggregate bandwidths only up to 570 Mbps. Increasing the number of clients further does not increase aggregate bandwidth because the host CPUs become saturated beyond 20 BitTorrent clients per machine.

In the undilated configuration, CPU becomes a bottleneck before network capacity. Next we use time dilation to scale CPU resources to provide additional processing for the clients without changing the perceived network characteristics. To scale CPU resources, we repeat the previous experiment but with VMs *dilated* with a TDF of 10. To keep the network capacity the same as before, we restrict the physical capacity of each client host link to 100 Mbps so that the underlying network appears as a 1-Gbps network to the dilated VMs. In effect, we dilate time to produce a new configuration with hosts with 10 times the CPU resources compared with the base configuration, interconnected by an equivalent network. Figure 12(b) shows the results of 10 runs of this experiment. With the increase in CPU resources for the clients, the BitTorrent overlay achieves close to the maximum 1-Gbps aggregate bisection bandwidth of the network. Note that the download times (in the dilated time frame) also improve as a result; due to dilation, though, the experiment takes longer in wall clock time (the most noticeable cost of dilation).

In the second configuration, network capacity now limits BitTorrent throughput. When using time dilation in the second configuration, we constrained the physical links to 100 Mbps so that the network had equivalent performance as the base configuration. In our last experiment, we increase *both* CPU resources and network capacity to scale the BitTorrent evaluation further. We continue dilating the VMs with a TDF of 10, but now remove the constraints on the network: client host physical links are 1 Gbps again, with a maximum aggregate bisection bandwidth of 10 Gbps in the dilated time frame. In effect, we dilate time to produce a configuration with 10 times the CPU and network resources as the base physical configuration.

Figure 12(c) shows the results of this last experiment. From these results, we see that the "faster" network leads to a significant decrease in download times (in the dilated time frame). Second, beyond 200 clients we see the aggregate bandwidth leveling out, indicating that we are again running into a bottleneck. On inspection, at that point we find that the end hosts are saturating their CPUs again as with the base configuration. Note, however, that in this case the peak bisection bandwidth exceeds 4 Gbps — performance that cannot be achieved natively with the given hardware.

Based upon these experiments, our results suggest that time dilation is a valuable tool for evaluating large scale distributed systems by creating resource-rich environments. Further exploration with other applications remains future work.

5 Related work

Perhaps the work closest to ours in spirit is SHRiNK [21]. SHRiNK reduces the overhead of simulating large-scale networks by feeding a reduced sample of the original traffic into a smaller-scale network replica. The authors use this technique to predict properties such as the average queueing delays and drop probabilities. They argue that this is possible for TCP-like flows and a network controlled by active queue management schemes such as RED. Compared to this effort, time dilation focuses on speed rather than size and supports unmodified applications.

The idea of changing the flow of time to explore faster networks is not a new one. Network simulators [3, 22, 25] use a similar idea; they run the network in virtual time, independent of wall-clock time. This allows network simulators to explore arbitrarily fast or long network pipes, but the accuracy of the experiments depends on the fidelity of the simulated code to the actual implementation. Complete machine simulators such as SimOS [24] and specialized device simulators such as DiskSim have also been proposed for emulating and evaluating operating systems on future hardware. In contrast, time dilation combines the flexibility to explore future network configurations with the ability to run real-world applications on unmodified operating systems and protocol stacks.

6 Conclusion

Researchers spend a great deal of effort speculating about the impacts of various technology trends. Indeed, the systems community is frequently concerned with questions of scale: what happens to a system when bandwidth increases by X, latency by Y, CPU speed by Z, etc. One challenge to addressing such questions is the cost or availability of emerging hardware technologies. Experimenting at scale with communication or computing technologies that are either not yet available or prohibitively expensive is a significant limitation to understanding interactions of existing and emerging technologies.

Time dilation enables empirical evaluation at speeds and capacities not currently available from production hardware. In particular, we show that time dilation enables faithful emulation of network links several orders of magnitude greater than physically feasible on commodity hardware. Further, we are able to independently scale CPU and network bandwidth, allowing researchers to experiment with radically new balance points in computation to communication ratios of new technologies.

Acknowledgments

We would like to thank Charles Killian for his assistance with setting up ModelNet experiments, Kashi Vishwanath for volunteering to use time dilation for his research and Stefan Savage and all the anonymous reviewers for their valuable comments and suggestions. Special thanks to our shepherd, Jennifer Rexford, for her feedback and guidance in preparing the camera-ready version. This research was supported in part by the National Science Foundation under CyberTrust Grant No. CNS-0433668 and the UCSD Center for Networked Systems.

References

[1] `http://bittorrent.com`.

[2] Linux TCP tuning guide. `http://www-didc.lbl.gov/TCP-tuning/linux.html`. Last accessed 03/25/2006.

[3] The network simulator - ns-2. `http://www.isi.edu/nsnam/ns/`. Last accessed 3/13/2006.

[4] Teragrid. `http://www.teragrid.org/`. Last accessed 03/25/2006.

[5] `tcpdump/libpcap`. `http://www.tcpdump.org`. Last accessed 03/25/2006.

[6] AMD. Amd64 secure virtual machine architecture reference manual. `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/33047.pdf`. Last accessed 3/13/2006.

[7] 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 SOSP* (2003), ACM Press, pp. 164–177.

[8] BUCY, J. S., GANGER, G. R., AND CONTRIBUTORS. The DiskSim Simulation Environment. `http://www.pdl.cmu.edu/DiskSim/index.html`. Last accessed 3/13/2006.

[9] COHEN, B. Incentives Build Robustness in BitTorrent. *Workshop on Economics of Peer-to-Peer Systems* (2003).

[10] DUDA, K. J., AND CHERITON, D. R. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the 17th ACM SOSP* (New York, NY, USA, 1999), ACM Press, pp. 261–276.

[11] FLOYD, S. High Speed TCP for Large Congestion Windows. `http://www.icir.org/floyd/papers/rfc3649.txt`, 2003. RFC 3649.

[12] FOSTER, I., KESSELMAN, C., NICK, J., AND TUECKE, S. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. `http://www.globus.org/alliance/publications/papers/ogsa.pdf`, January 2002. Last accessed 03/29/2006.

[13] INTEL. Intel virtualization technology. `http://www.intel.com/technology/computing/vptech/index.htm`. Last accessed 3/13/2006.

[14] JACOBSON, V., BRADEN, R., AND BORMAN, D. TCP Extensions for High Performance. `http://www.rfc-editor.org/rfc/rfc1323.txt`, 1992. RFC 1323.

[15] JAIN, R. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991. Chapter 12.

[16] KATABI, D., HANDLEY, M., AND ROHRS, C. Congestion control for high bandwidth-delay product networks. In *SIGCOMM* (2002), ACM Press, pp. 89–102.

[17] KELLY, T. Scalable TCP: improving performance in highspeed wide area networks. *SIGCOMM Comput. Commun. Rev. 33*, 2 (2003), 83–91.

[18] LAKSHMAN, T. V., AND MADHOW, U. The performance of TCP/IP for networks with high bandwidth-delay products and random loss. *IEEE/ACM Trans. Netw. 5*, 3 (1997), 336–350.

[19] LOVE, R. *Linux Kernel Development*. Novell Press, 2005.

[20] MOGUL, J. C. TCP offload is a dumb idea whose time has come. In *9th Workshop on Hot Topics in Operating Systems* (2003), USENIX.

[21] PAN, R., PRABHAKAR, B., PSOUNIS, K., AND WISCHIK, D. SHRiNK: A method for scaleable performance prediction and efficient network simulation. In *Proceedings of IEEE INFOCOM* (2003).

[22] RILEY, G. F. The Georgia Tech Network Simulator. In *MoMeTools '03: Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research* (2003), pp. 5–12.

[23] RIZZO, L. Dummynet: A simple approach to the evaluation of network protocols. *SIGCOMM Comput. Commun. Rev. 27*, 1 (1997), 31–41.

[24] ROSENBLUM, M., BUGNION, E., DEVINE, S., AND HERROD, S. A. Using the SimOS machine simulator to study complex computer systems. *ACM Trans. Model. Comput. Simul. 7*, 1 (1997), 78–103.

[25] SZYMANSKI, B. K., SAIFEE, A., SASTRY, A., LIU, Y., AND MADNANI, K. Genesis: A System for Large-scale Parallel Network Simulation. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation (PADS)* (May 2002).

[26] VAHDAT, A., YOCUM, K., WALSH, K., MAHADEVAN, P., KOSTIC, D., CHASE, J., AND BECKER, D. Scalability and accuracy in a large-scale network emulator. *SIGOPS Oper. Syst. Rev. 36* (2002), 271–284.

[27] VMWARE. Timekeeping in VMWare Virtual Machines. `http://www.vmware.com/pdf/vmware_timekeeping.pdf`. Last accessed 03/24/2006.

[28] WARKHEDE, P., SURIAND, S., AND VARGHESE, G. Fast packet classification for two-dimensional conflict-free filters. In *Proceedings of IEEE INFOCOM* (July 2001).

[29] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Scale and performance in the Denali isolation kernel. *SIGOPS Oper. Syst. Rev. 36* (2002), 195–209.

[30] XU, L., HARFOUSH, K., AND RHEE, I. Binary increase congestion control (BiC) for fast long-distance networks. In *Proceedings of IEEE INFOCOM* (2004).