Mininet Performance Fidelity Benchmarks

Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, Nick McKeown

October 21, 2012

1 Introduction

This initial Mininet technical report evaluates the performance fidelity of the Mininet/Mininet-HiFi system by examining results from two classes of experiments: link tests (section 2) and microbenchmarks (section 3.)

This is a work in progress which will be updated as we develop more tests as well as a deeper understanding of the performance and timing characteristics of Mininet/Mininet-HiFi and Container-Based Emulation techniques.

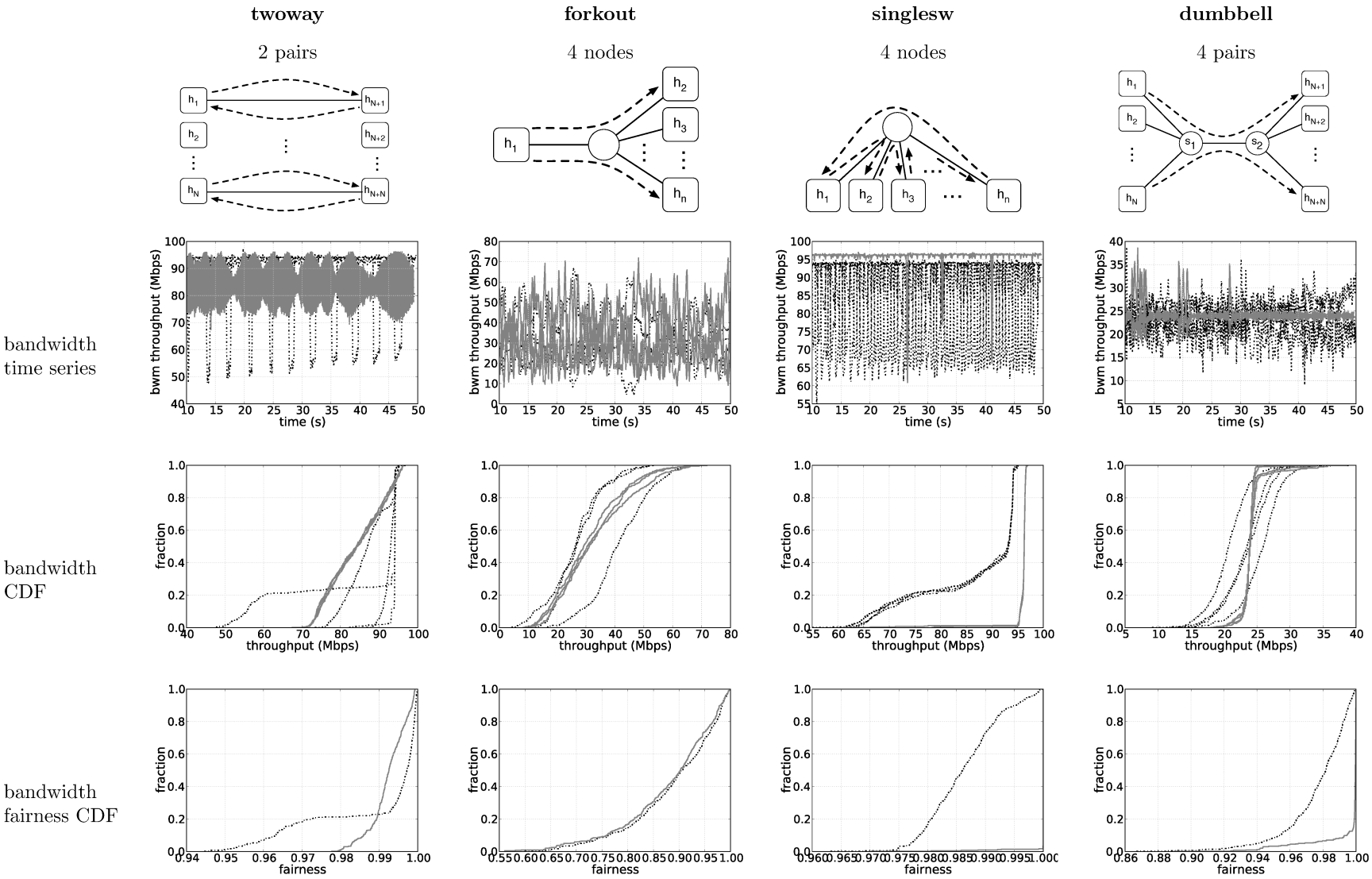
2 Validating Mininet-HiFi

We evaluate whether Mininet-HiFi, with its resource isolation mechanisms, can faithfully emulate a complete network. How close do the results come to hardware when running more complex code and a full network topology, rather than a single link?

Our approach is to run tests on simple networks with multiple TCP flows, then compare the bandwidths achieved to those measured on a testbed with an equivalent topology. For each test, we know roughly what to expect - multiple flows sharing a bottleneck should get an equal share - but we do not know the variation between flows, or whether TCP will yield equivalent short-term dynamics.

The short answer is that Mininet-HiFi yields similar TCP results, but with greater repeatability and consistency between hosts when compared to hardware. However, for one simple test topology, a dumbbell with UDP traffic, it produces the wrong result; we describe this case and our implemented fix later in this section.

Table 1: **Validation Tests**.
Each graph shows measured interface counters for all flows in that test, comparing data from Mininet-HiFi running on a dual-core Core2 1.86GHz, 2GB machine against a hardware testbed with eight machines of identical specifications. In many cases the individual CDF lines for Mininet-HiFi results overlap.
Legend: **Mininet-HiFi: solid red lines.** **Testbed: dashed black lines.**



2.1 Testbed

Our testbed includes eight identical dual-core Core2 1.86 GHz, 2 GB RAM Linux servers. These eight machines are connected as dumbbell with four hosts on each end through an NEC PF5240 switch. The switch is split into two VLANs and connected with a physical loopback cable. Each server has two motherboard ports, and we configure one for remote access and the other to the isolated test network (the dumbbell). When showing Mininet-HiFi results in this section, we use a single dual-core PC from within the testbed. For both hardware and software, all links are set to 100 Mbps full-duplex.

Note that our hardware will not match other testbeds, and in fact, we may not be able to measure our own testbed with enough fidelity to accurately re-create it in Mininet-HiFi. For example, we don't know the amount of buffers in the switch or its internal organization. Also, without a hardware NIC across a bus, we expect much less variability from the NIC and a noticeably lower RTT between hosts on the same switch. Hence, one would not expect the results to match exactly.

2.2 Tests

Table 1 presents our validation results. Each column corresponds to an experiment with a unique topology and configuration of flows. Each row corresponds to a different way to present the same bandwidth data, as a time series, CDF, or CDF of fairness. To measure fairness between flows we use Jain's Fairness Index, where $1/n$ is the worst possible and 1 represents identical bandwidths. Mininet-HiFi results are highlighted in solid red lines, while testbed results are dashed black lines. In the time series and bandwidth plots, each TCP Reno flow has its own line.

Two-Way Test. The goal of this test is to verify whether the links are "truly duplex" and whether the rate-limiters for traffic flowing in either direction are isolated. The network consists of two two-host pairs, each sending to the other. Compared to hardware, this is the one case where Mininet-HiFi shows consistently more variation over time, however each host has the exact same distribution of bandwidth over time; all four Mininet-HiFi lines overlap.

Fork-Out Test. The goal of this test is to verify if multiple flows originating from the same virtual host (container) get a fair share of the outgoing link bandwidth, or if they are incorrectly affected by the CPU scheduler. The test topology consists of four hosts, connected by a single switch, with one sender and three receivers. The bandwidths received by the three Mininet-HiFi hosts are squarely within the envelope of the hardware results, and the fairness CDF is nearly identical.

Single-Switch Test. The goal of this test is to see what happens with potentially more-complex ACK interactions. The single-switch test topology consists of four hosts, each sending to their neighbor, with the same total bandwidth requirement as the Two-Way test. The Mininet-HiFi results show less variation here than the hardware testbed.

Dumbbell Test. The goal of this test is to verify if multiple flows going over a bottleneck link get their fair share of the bandwidth. The dumbbell test topology consists of two switches connected by a link, with four hosts at each end. Four hosts each send to the opposite side. We see throughput variation in the TCP flows, due to ACKs of a flow in one direction contending for the queue with the reverse flow, increasing ACK delivery delay variability. The median throughput is identical between Mininet-HiFi and hardware, but again the Mininet results show less variability over time and between hosts.

In general, Mininet-HiFi shows less variation between hosts with TCP flows than the testbed. Results on Mininet-HiFi were decidedly more consistent. For example, the repeating pattern seen in the Two-Way test occasionally appeared on hardware, but never on Mininet-HiFi. However, as we see next, putting the resource-isolation pieces together is not enough.

2.3 UDP Synchronization Problem + Fix

When running UDP tests on the dumbbell topology, we noticed that the bandwidth was not getting equally shared among the flows. We found that one flow was receiving all the bandwidth, while the others starved. Repeating the test with two pairs of hosts produced the same result; just one flow received all the bandwidth.

The reason for this behavior is that all the emulated nodes in the system operate under a single system clock domain. This perfect synchronization leads the UDP streams to become perfectly interleaved at the bottleneck interface. When there was room for a packet in the bottleneck queue, a packet from the first flow would always take the slot, and the next packet from each other flow would be dropped. While not technically a bug in the queue implementation, this behavior does prevent us from getting results that we would expect to obtain on real networks. In real networks, hosts, switches, and switch interfaces all operate on different clock domains. This leads to an inherent asynchrony, preventing pathological scenarios like the one observed. The continuing drift of one interface relative to the other yields fairness over time.

The single-clock domain sync problem is not unique to our emulator; it also appears in simulators, where the typical solution is to randomly jitter the arrival of each packet event so that events arriving at the same virtual time get processed in a random order. Since our emulator only knows real time, we use the following algorithm to emulate the asynchrony of real systems and jittered simulators on Mininet-HiFi:

- On each packet arrival, if the queue is full, temporarily store it in an overflow buffer (instead of dropping it, as a droptail queue would).
- On each dequeue event, fill the empty queue slot with a packet randomly picked from the overflow buffer (instead of the first packet), and drop the rest.

After implementing the jitter-emulation solution, we observed the flows achieving fairness close to that measured on real hardware, similar to the TCP case. We saw similar synchronization with a Fork-In test with two hosts sending to one host, except instead of complete starvation, the bandwidth was arbitrarily split between the flows, with a split ratio that remains consistent for the duration of the experiment. The modified link scheduler code fixed this bug, too.

In the next section we see how the observed fairness properties of Mininet-HiFi enable it to reproduce results measured in hardware from other successful research projects.

Sched	vhosts	goal cpu%	mean cpu%	maxerr cpu%	rmserr %
bwc	50	4.00	4.00	0.41	2.1
	100	2.00	2.00	0.40	2.9
rt	50	4.00	4.00	0.39	1.9
	100	2.00	2.00	0.40	3.2
default	50	4.00	7.98	4.44	100.0
	100	2.00	3.99	2.39	99.6

Table 2: Comparison of schedulers for different numbers of *vhosts*. Target is 50% total utilization of a four core CPU.

3 Micro-benchmarks for CPU and Link Schedulers

CBE relies on the accuracy and correctness of the CPU scheduler and the link scheduler. The following micro-benchmarks explore the fidelity of each scheduler in isolation. For each scheduler, we measure how precisely it is scheduled over time (i.e. the CPU utilization during each second for each process; the data-rate of each link), and the consequences of serialization scheduling epochs (i.e. the time from when a process should be scheduled, until the time it is; the time from when a packet should depart, until the time it does).

The micro-benchmarks presented here are from the Mininet-HiFi distribution, run on an Intel i7 CPU with 4 cores running at 3.20GHz with 12GB of RAM.

3.1 The CPU Scheduler

The time a process is scheduled each second: Table 2 compares the CPU allocation we *want* a *vhost* to receive against the *actual* CPU time given to it, measured once per second. The *vhosts* are simple time-waster processes, and the total CPU utilization is 50%.

Consider the first row: The *bwc* scheduler allocates each of the 50 *vhosts* 4% of a CPU core (there are four cores, so total CPU utilization is 50%). Each second we measure the amount of time each process was *actually* scheduled and compare it with the target. The maximum deviation was 0.41% of a CPU (i.e. in the worst-case second, one process received 3.59% instead of 4%). The rms error was 2% of the target (i.e. 0.008% of a CPU core within a second). The results are similar for 100 *vhosts* and for the *rt* scheduler. In other words, the *bwc* and *rt* schedulers both give the desired mean allocation, *during each second*, within a small margin of error. As expected, in both cases, the *default* scheduler gives a very different allocation, because it makes no attempt to limit the per-process CPU usage.

The results in Table 2 are limited by our measurement infrastructure, which has a resolution of only 10ms – which for 200+ *vhosts* is not much longer than each process is scheduled. This prevented us from making reliable per-second measurements for larger numbers of *vhosts*.

3.2 CPU Delay

The second set of benchmarks measures if *processes are scheduled at precisely the right time.* This is important for applications that need to perform tasks periodically, e.g. a video streaming application sending packets at a specific rate.

We conduct two tests. In each case, *vhost-1* sets an alarm to fire after time t, and we compare the actual versus desired waking time. $N-1$ *vhost* processes run time-waster processes. In Test 1 *vhost-1* is otherwise idle. In Test 2 *vhost-1* also runs a time-waster process to create "foreground load".

Figure 1 shows the relative timer error in the two tests for various values of N, t, and for the two schedulers - *cbw* and *rt*. Both introduce small errors in the timer for Test 1, increasing as t approaches the scheduling period (10ms), where it may fall either side of the boundary. But the main takeaway is that *rt* fails to schedule accurately in Test 2. We discounted *rt* from further consideration.

3.3 The Link Scheduler

We created micro-benchmarks to test the accuracy to which the OS can emulate the behavior of a link, for both TCP and UDP flows.

Throughput: We measure the throughput of *vhost* A transmitting to *vhost* B over a `veth` pair using TCP. Figures 2(a) and 2(b) show the distribution of attained rates (normalized) over a 10ms time interval, for the `htb` and `hfsc` schedulers, respectively. Both link schedulers constrain the link rate as expected, although the variation grows as we approach 1Gb/s. The table highlights the need to run micro-benchmarks as part of the process of determining whether an experiment can be accurately run on a given computer, and for mapping *vhosts* to CPU cores.[1]

Our next test runs *many* `veth` pairs in parallel, to find the point at which the CPU can no longer keep up, and to where the CPU cycles are spent for switching data through the kernel. Figure 3(a) shows the total time spent by the kernel on behalf of end hosts, including system calls, TCP/IP stack, and any processing by the kernel in the context of the end host. The system time is proportional to the number of links, for a given link rate (10Mb/s - 1Gb/s). The graph shows where the system time is insufficient to keep up (where the points lie below the proportionality line), and the performance fidelity breaks down.

Figure 3(a) does not include processing incurred by the network switches, which are accounted as `softirq` time; these are shown in Figure 3(b). As expected, the system time is proportional to the *aggregate* switching activity. Should a measured point deviate from the line, it means the system is overloaded and falling behind.

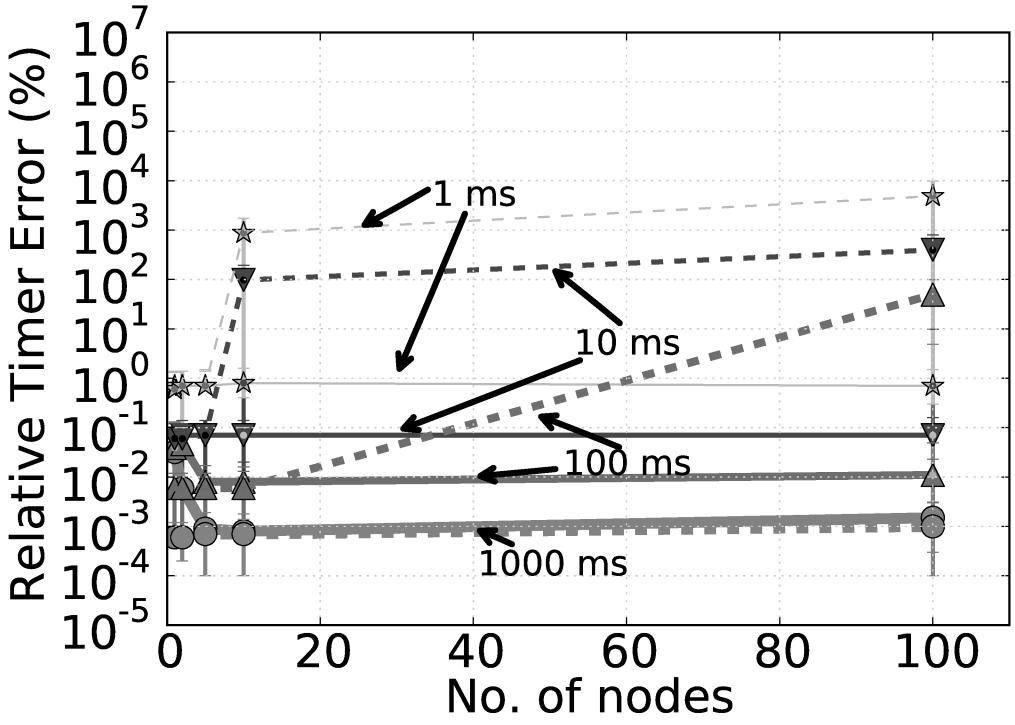
We can now estimate the total number of `veth` links that we can confidently emulate on our server. If we connect L links at rate R between a pair of *vhosts*, we expect a total switching activity of $2LR$. Figure 4(a) shows the *measured* aggregate switching capacity for $L = 1 - 128$ `veth` links carrying TCP, at rates $R = 10Mb/s - 1Gb/s$. The solid lines are the ideal aggregate switching activity ($2LR$) when there is sufficient CPU resource. Measured data points below the line represent "breaking points" when the system can no longer yield high fidelity results; as expected, we see that fidelity is lost as we push the CPU utilization to 100%. We recommend profiling a system before running research experiments, to understand its limits.

3.4 Round-trip latency effects

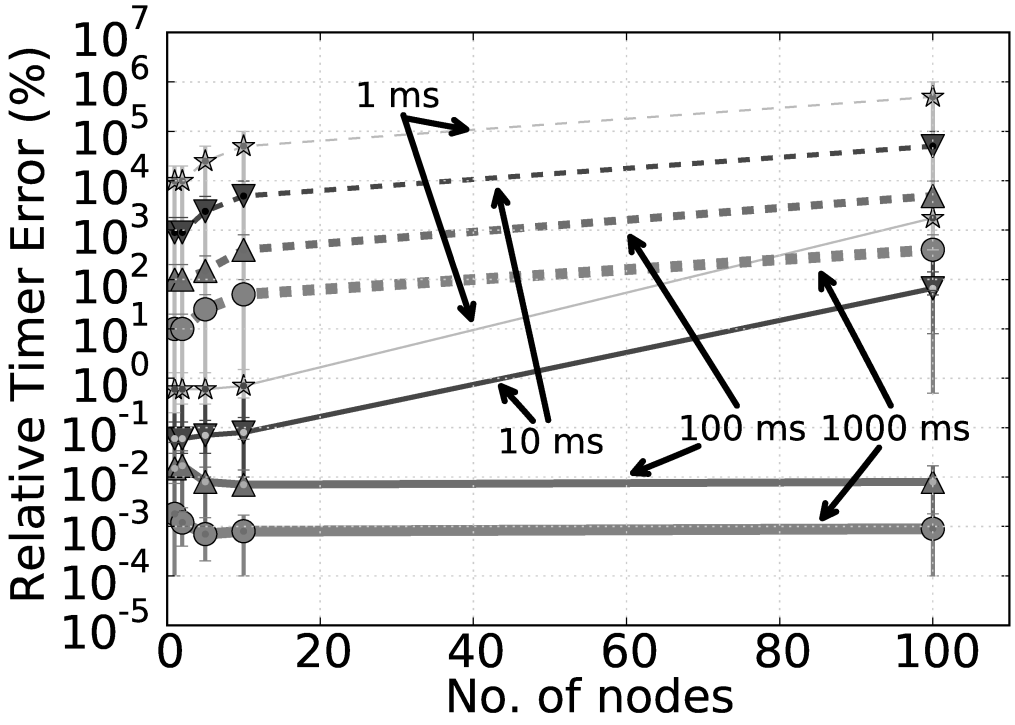
To examine the effects of background load on round-trip latency, we ran a series of experiments where a client and server are communicating in an RPC-like scenario: the client sends a packet to the server which then sends a reply, while other hosts within the experiment run CPU-intensive programs. Because I/O-bound processes will have a lower overall runtime than CPU-bound processes, Linux's default CFS scheduler automatically prioritizes them over CPU-bound processes, waking them up as soon as they become runnable. We wished to verify that CFS bandwidth limiting preserved this behavior when the hosts were limited to 50% of overall system bandwidth.

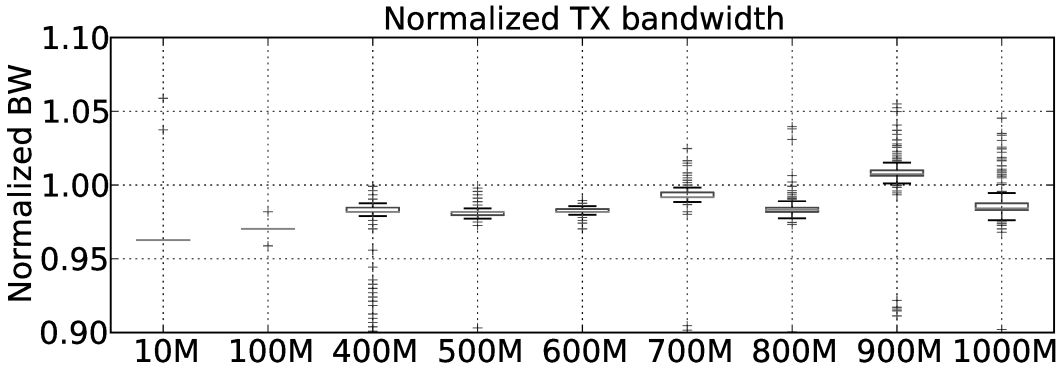
Figure 5 shows that although the average request-response time does not increase dramatically from 2 to 80 hosts, the variation (shown by error bars of +/- twice the standard deviation, corresponding to a 95% confidence interval) is larger at higher levels of multiplexing an dbackground load. This indicates that it is more likely that a client or server which is ready to run may have to wait until another process completes its time slice.

[1]The benchmark revealed a bug in the link scheduler, which remains to be resolved. At rates close to 1Gb/s, it appears we must read from a buffer about 20Mb/s faster than we write to it. We believe this is a bug, and not a fundamental limitation.

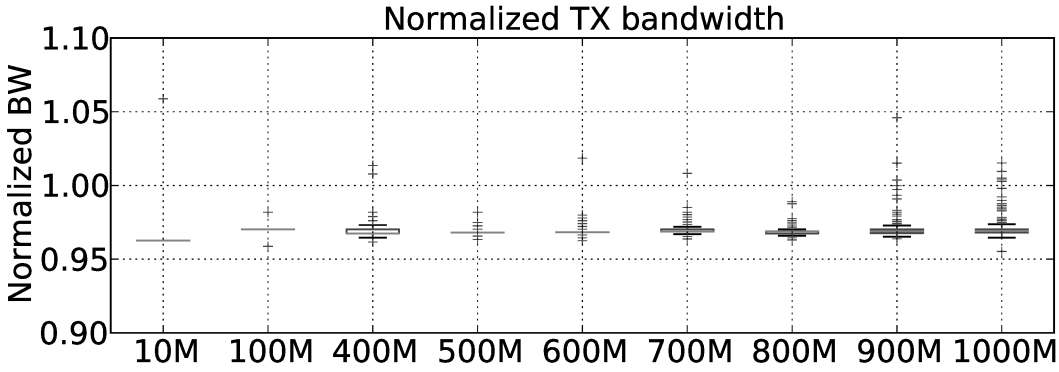


(a) Timer fidelity: *cbw*



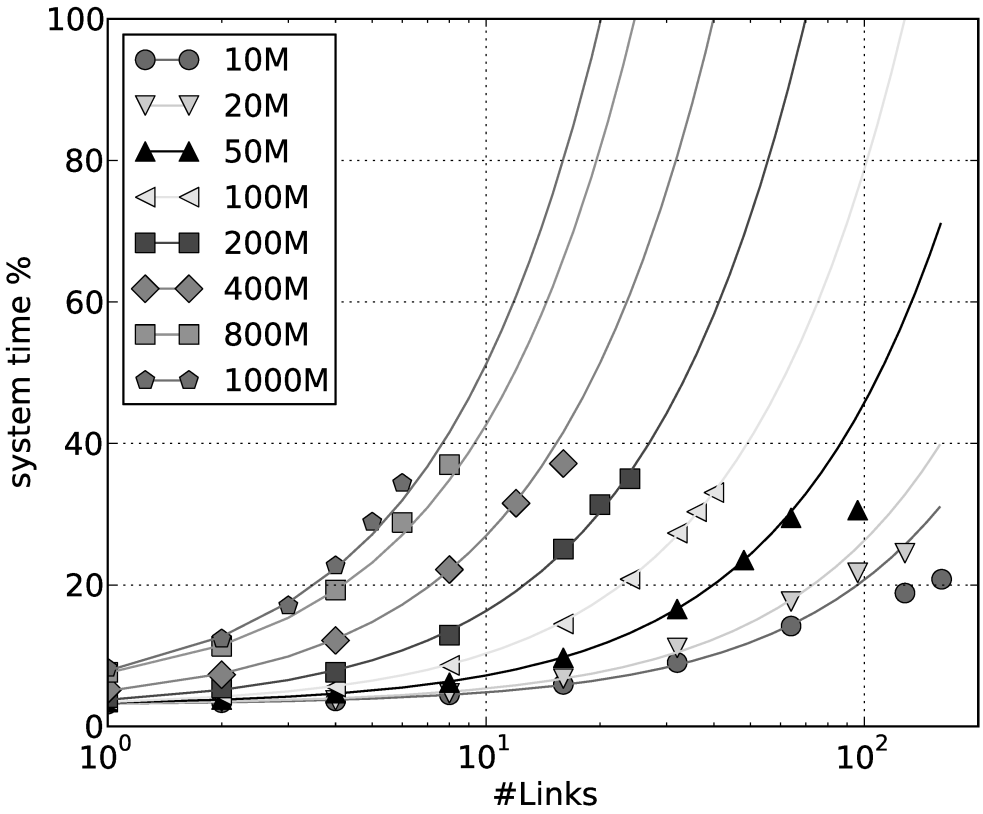


(a) Rates using HTB

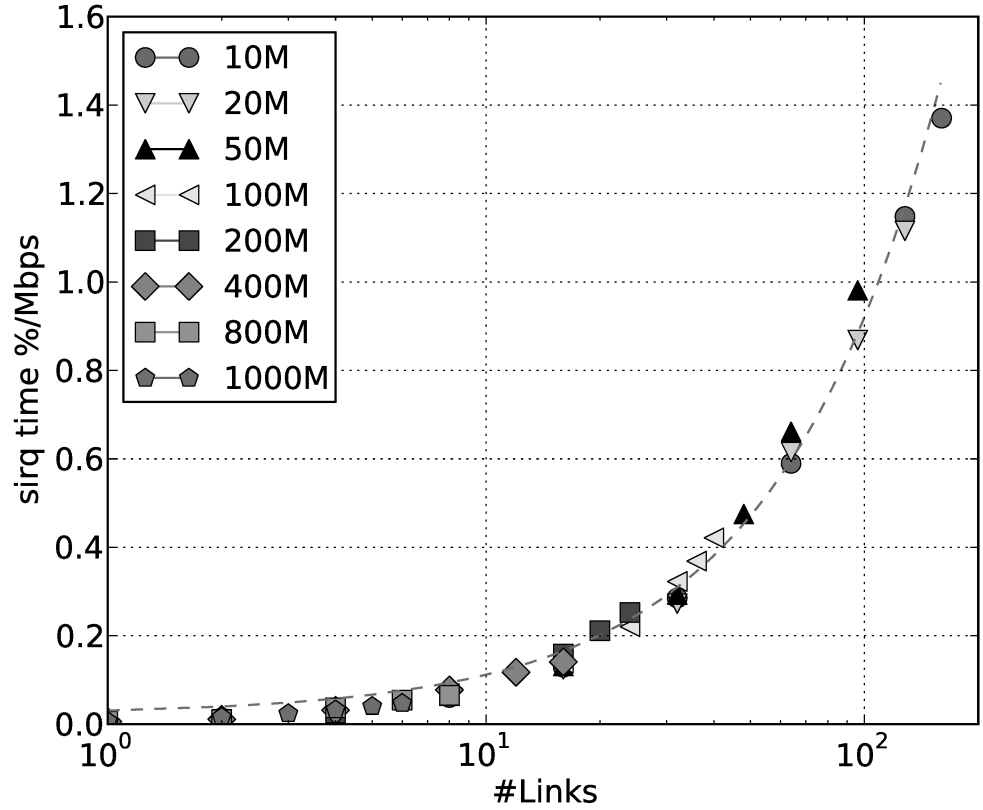


(b) Rates using HFSC

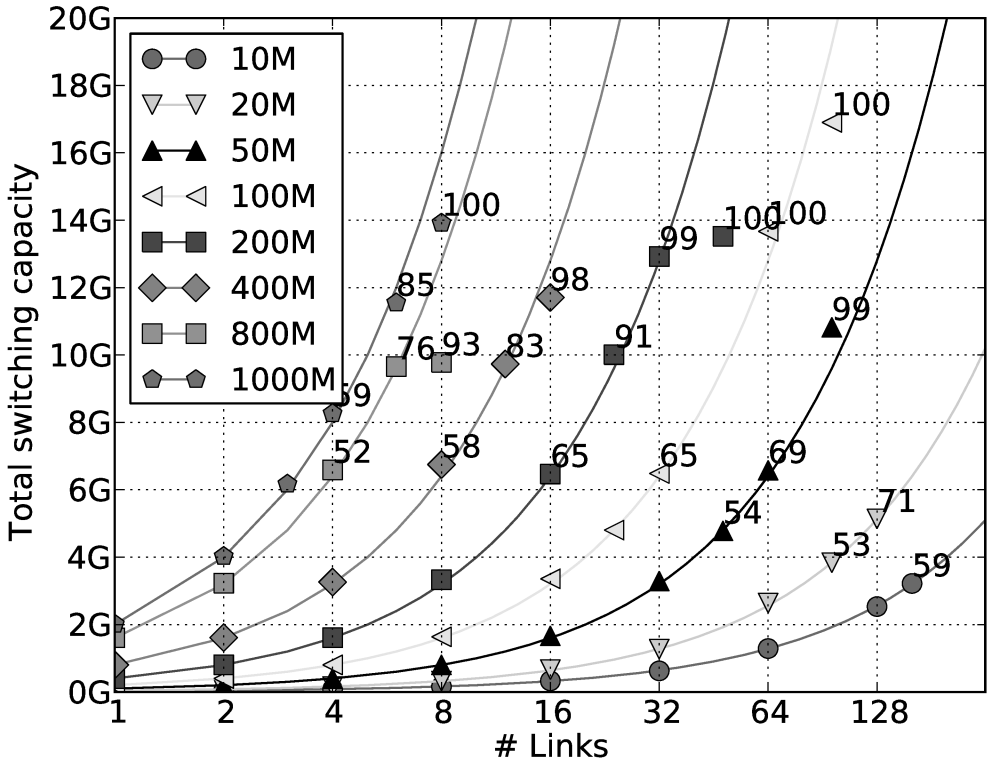
Figure 2: Rates for a single `htb` & `hfsc`-scheduled link, fed by a TCP stream.



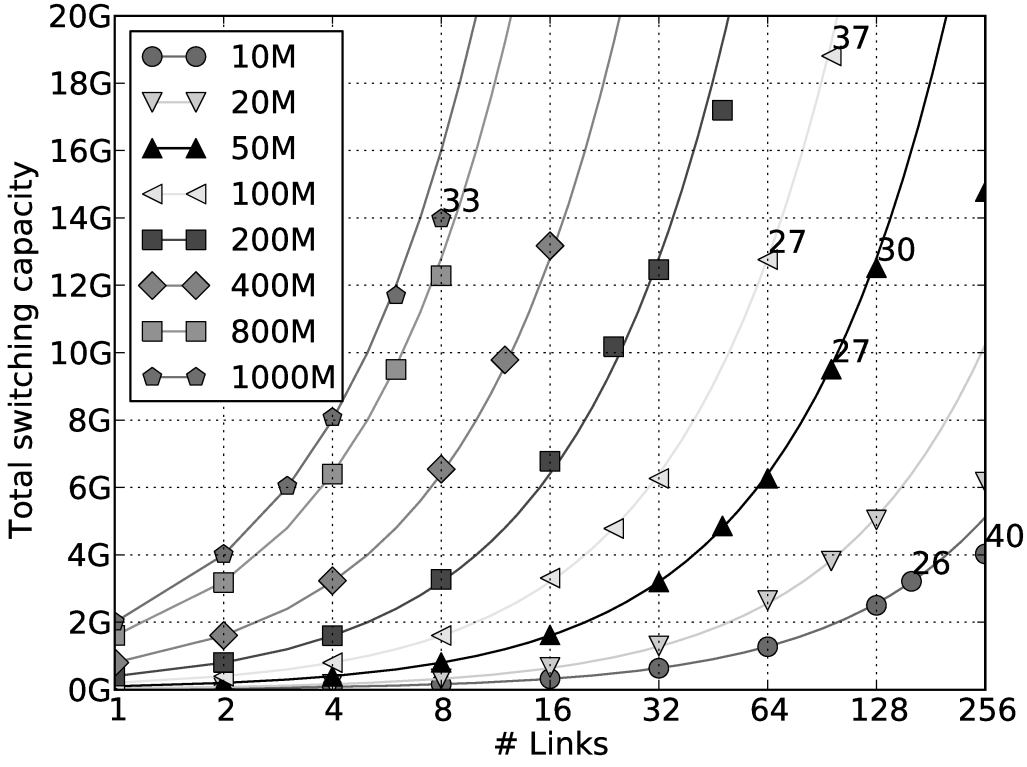
(a) Kernel CPU usage for various rates and number of links



(b) SoftIRQ CPU per Mbps for various rates and number of links



(a) Multiple links in parallel



(b) Multiple links in series

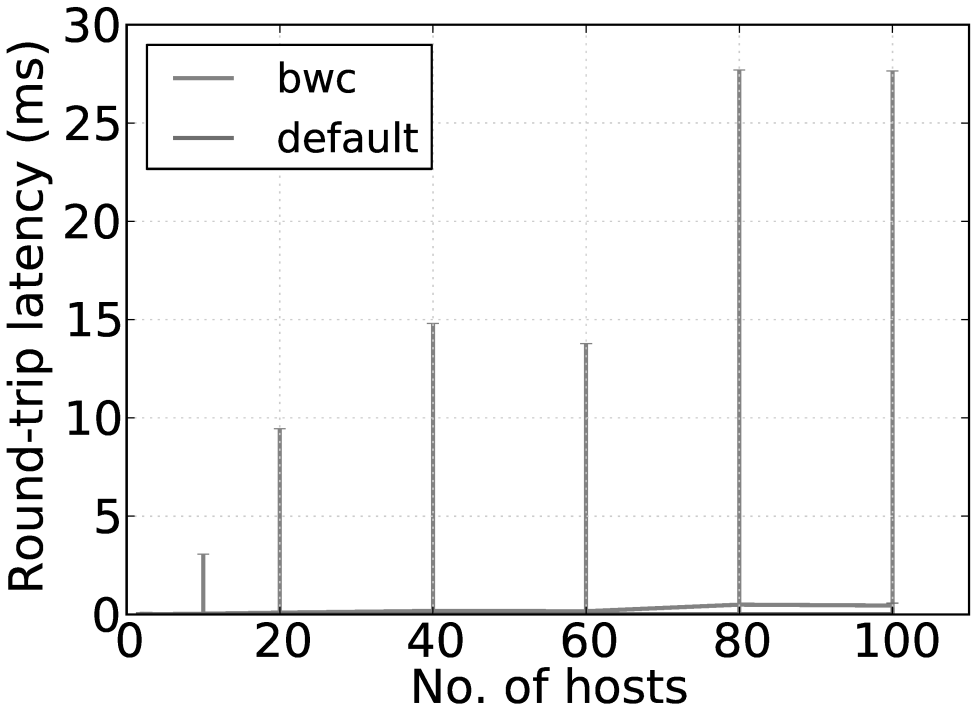


Figure 5: Round-trip latency in the presence of background load.