Accurate Emulation of Fast Optical Circuit Switches

Henrique Rodrigues, Richard Strong, Tajana Rosing
University of California, San Diego

***Abstract*—Fast All-Optical Circuit Switches (AOCS) are emerging as a solution to high bandwidth demands of computer networks. Unlike traditional AOCSs, which use has been restricted to provide static point-to-point circuits due to their slow circuit reconfiguration delays, recent fast AOCS prototypes are able to multiplex circuits in μs timescales. Fast AOCS switching reduces the gap between Electronic Packet Switching (EPS) and AOCSs, as one could use very short-lived circuits to multiplex packets without the need of Optical-Electrical-Optical (OEO) conversions. This could lead to drastic reductions in network power consumption and transceiver-related costs. However, studying application performance using fast AOCS, as well as exploring efficient scheduling schemes for such AOCSs, is restricted to few research groups which can build fast AOCS prototypes. To address this problem we designed OCSEMU, a distributed emulation platform that mimics the behavior of fast OCSs using a traditional EPS. We show that OCSEMU achieves comparable performance to the state-of-the-art AOCS prototype, providing the extensibility necessary to explore application performance and optimizations of emerging AOCSs.**

Keywords*—*Optical Switching, Emulation, Circuit Scheduling, Data Center Switch

I. Introduction

Electronic packet switches (EPS) have been the basic building block for large data center networks. However, scaling bandwidth capacity of EPS interconnects incurs significant rise in power consumption, costs, and maintenance [1], [2]. In this scenario, All Optical Circuit Switches (AOCS or simply OCS) are gaining attention as an alternative to EPS networks [3], [4], [1], [5]. The technology provides abundant bandwidth per port through Wavelength Division Multiplexing (WDM) at minimal power consumption, and is expected to reduce equipment and operational costs of high bandwidth interconnects [4].

Despite OCS advantages and recent research progress, little is known about application performance on newer OCS interconnects. Optical switches still have some physical and design limitations that could impact application performance negatively. First, current OCSs cannot rely on packet headers for switching decisions, and an external controller must dictate when and for how long circuits should be setup. Second, optical buffering is not as efficient as digital buffers, restricting OCS forwarding *flexibility* as the switch cannot absorb bursts of traffic or sudden changes in demand. Most importantly, circuit switching might incur significant *reconfiguration delay*, as setting up a circuit often involves controller communication and movement of mechanical components, during which no traffic can be carried by the OCS [3].

These problems are well known, and led to promising research progress on emerging fast OCSs that provide *reconfiguration delays* of a few microseconds [6], [7]. However, studies on application performance using state-of-the-art hardware prototypes are restricted to synthetic and mostly static traffic patterns [1], [5]. On the other hand, actual distributed applications are complex, composed of multiple layers of software and hardware modules that interact towards a common goal. Data flows from distributed applications for instance are usually correlated, and delaying a single one of them might slow down the entire application [8]. Furthermore, performance is often sensitive to hardware and software tuning, involving CPU, memory, storage and distributed load balancing. Network Interface Card (NIC) offloading features, for example, can change network performance dramatically and change the times in which packets are transmitted (See Sec. V-A).

Research on fast OCS designs is in its early stages [6], [7]. Therefore, experimentation with real applications would require building OCS prototypes, which is unfeasible for multiple research groups and industry teams. Restricted access to newer optical switching technologies also contributes to slow progress of research in the area. An alternative is to simulate distributed applications and networks. However, this is not scalable even for a small-sized cluster [9]. The level of detail of simulation is often limited, possibly lacking important interactions between applications, Operating Systems (OS), network stack and the actual network devices. We take a different approach to help accelerate research progress in the area and propose an emulation testbed for fast OCS.

In this paper we present and validate OCSEMU, an OCS network emulation environment that mimic the behavior of emerging fast circuit switches using traditional servers and EPSs. As an emulation platform, it allows operators and researchers to deploy and benchmark the same applications and protocols on the emulated network as they would on real hardware. OCSEMU operates as a pluggable module to the OS and requires minor changes to existing network drivers. The OS network stack is left unchanged, allowing further experimentation with different versions of Layer 2/3 protocols.

II. Related Work

Optical Circuit Switch use as a "dynamic interconnect" envisioned offloading *hotspots* of traffic from a congested EPS to an OCS network [4], [10], [3]. Prototypes show that OCSs can offer high bandwidth capacity at low power, providing non-oversubscribed network capacity at lower cost. However, traditional designs using 3D-MEMS OCSs are not efficient for short-lived dynamic traffic patterns [4], and OCS technology is evolving towards faster and more flexible photonics switching [6], [7]. To the best of our knowledge, there are only two Optical Circuit Switch designs which can offer μs switching times with reasonable port counts [6], [7]. Application performance on such new architectures is mostly unexplored. OCSEMU provides a testbed for exploration of these and future OCS designs via emulation.

Application performance on networks using OCSs is studied in c-Through [10], but using slow (millisecond) OCSs. More recently, Kapoor et. al. [11] explored NIC packet transmission behavior at microsecond time scales, showing that fast OCSs might be suitable to carry various network flows.

Network emulation offers multiple advantages over pure simulation tools such as NS-2/3 [12], [13]. However, emulation platforms such as NIST Net, NetEm or DummyNet [14], [13] focus mainly on modeling single link characteristics, either by introducing delays or packet reordering/drops. The approach has been used to study protocols on packet switched networks (e.g. TCP variants). Instead, OCSEMU focuses on distributed coordination of multiple links to emulate circuits being setup and torn down in a few μs. Although emulation tools for OCSs exist [15], they model slow OCSs and use control protocols such as OpenFlow or RSVP, which are too slow to reconfigure circuits in μs scales [4], [10]. Compared to prior solutions, OCSEMU uses single link scheduling algorithms similar to NetEm, but has a global coordination module for μs-scale distributed coordination of emulated circuits.

Perhaps the closest work to ours is *SliceTime* [9]. It uses virtual machines (VM) to manipulate the time perceived by applications running inside VMs. The idea is to slow down execution of applications, and pace their runtime to a slower network emulation. A similar strategy is used in DieCast [16]. We do not slow down applications. Instead, OCSEMU goes after the challenge of synchronizing the batched and non real time OS network stack's transmission (TX) and reception (RX) of data across servers, to mimic the expected behavior of a μs OCS interconnect. OCSEMU requires minimal changes to existing OS software, allowing experimentation of native applications running on physical servers as well as inside VMs.

c-Through [10] uses OCS emulation to reach similar conclusions as in Helios [4]. However, the proposed system was designed to use slow OCSs, with long control loops to gather statistics and reconfigure networks using VLANs. c-Through could use fast OCSs, but the algorithms to control the OCS are slow, and would not explore the potential of newer OCSs.

Software Defined Network (SDN) has some similarities to OCS, as the network control plane is decoupled from its data plane. Emulation environments for SDN have been proposed [12], [17], [15]. OCSEMU provides similar control for OCS environments, but it operates at much higher speeds (μs scale) than current SDN controllers (usually 10s of ms).

III. Background

Before describing OCSEMU, it is useful to outline some differences between an OCS and an EPS. First, unlike traditional packet switches, an OCS does not have local buffers. On the good side, no electronic packet buffering combined with WDM allows OCSs to carry traffic at $Tbps$ per port with minimal power consumption [4]. However, the lack of local buffers also means that the end-node data transmission needs to be synchronized with circuit availability. Losses will happen if transmission takes place when circuits are being setup, teared down or unavailable. Thus, end-nodes are responsible to buffer data demands for a particular circuit until it becomes available in the OCS data plane. Furthermore, lack of buffers in the OCS also means that traffic demand information is not a *bus* away, on the same silicon chip. The information needs to be collected from end-nodes. Depending on OCS efficiency, speed and workload characteristics, one could use an OCS to interconnect either servers in a rack or core/aggregation packet switches which interconnect sets of racks [4], [6]. End-nodes can thus be either server NICs or packet switches, as both use transceivers to convert digital data to optical signals.

Delays in demand collection can be mitigated using a fair round robin circuit schedule (similar to TDMA), as in the Mordia OCS [1]. Figure 1 depicts such scheme with 3 nodes connected to an OCS. Each circuit connects two nodes and gets a fixed amount of network resources over time. In the example with 3 nodes A, B and C, that would mean 2 network configurations, with circuits sets $\{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$ and $\{A \rightarrow C, B \rightarrow A, C \rightarrow B\}$ visited in sequence over time. The controller is responsible for reconfiguring the OCS optical data plane and coordinating transmission through a separate control plane[1]. We refer to OCS reconfiguration delays (i.e. switching) as *nights*, when no light can flow through the interconnect, and the periods of stable connectivity as *days*. This scheme avoids resource waste with a long term demand-aware scheduler, which removes or adds nodes to the schedule.

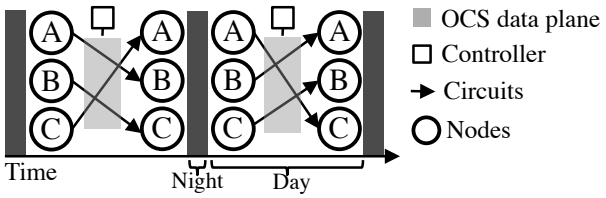


Fig. 1: Representation of OCS schedule

Another limitation is the absence of multicast in OCSs. This restricts their use to point-to-point (circuit) transmissions. However, as switching time decreases (current fast OCSs offer nights of a few μs), multiple circuits connecting distinct end-nodes can be established within a short time interval with a small duty cycle. For example, consider circuit multiplexing with 90 μs *days* and 10μs *nights*, giving us 90% of useful data transmission. In this case, if a single node A intends to send data to 10 other nodes, each receiver will get an equal share of network resources, and have a circuit from A available within 1 ms, with A requiring only $\sim$1 MB of total buffering [18].

IV. OCS Emulation Environment

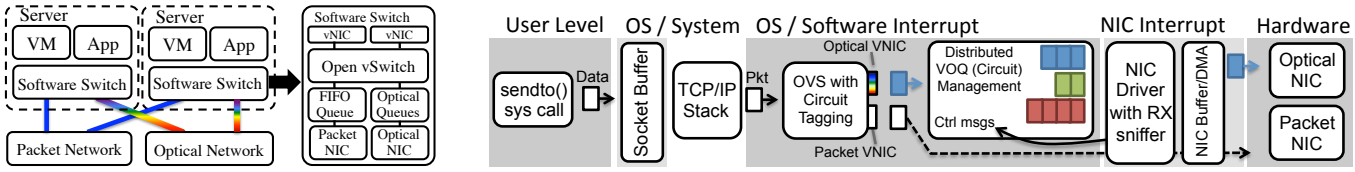
This section describes how we emulate the behavior of an OCS, as described above, with μs precision relying on software-based link schedulers and controller. The emulation environment operates mainly on servers, which in turn connect to a traditional EPS, as shown in Figure 2a. Each server has two network interfaces, one connected to the control plane, which carries coordination messages, and another one to the emulated optical data plane, which carries actual application data. We evaluate the emulation of an OCS with 23 ports connected to servers spread in two racks, as these are the resources available in our testbed. However, 23 is not a strict limit. OCSEMU is suitable for either larger or smaller number of servers, depending on the number of ports available in the EPS (e.g. the Huawei CE EPS [2] offers up to $1,536$ ports).

Larger applications that require much more servers can also be explored using virtual machines, because OCSEMU does not require significant changes to a traditional OS (i.e. "App" in Figure 2a can be either a VM or an application running on a server). Furthermore, hybrid packet-optical networks, in which data is carried by optical and electrical data planes (as in [5], [10]), can also be explored with this design, but here we focus only on the optical data plane emulation.

A. Kernel level circuit emulation

In each server, we use a software switch and fine-grain queue management for the server TX synchronization, which

[1]The control plane carries only control messages, and is not shown in the picture. This is similar to the design of SDN with control and data planes.



(a) Software and hardware used. Packet network is a 10G EPS and Optical is emulated over the EPS

(b) Virtual Switching and wavelength-based queue management added to Kernel and NIC driver for accurate OCS emulation. Control messages are sent by Synchronization Controller (Sec. IV-C)

Fig. 2: Details of software and hardware used in our experiments.

decides when traffic should be sent to the emulated optical data plane. Figure 2b gives an overview of a packet's life from an application request to the Network Interface Card (NIC) transfer. When an application sends data across the network, it goes through several levels in the OS until it is transmitted by the NIC. To have fine-grain control of all traffic sent on the network, we insert our code within and right before the NIC driver. We also isolate a single CPU core from the rest of the system, using it to perform network control. The Linux Kernel provides this feature through the *isolcpu* parameter, allowing us to have minimal intervention from other applications and ensure high responsiveness in network event handling.

The first change we introduce to a packet's life is the integration of Open vSwitch (OVS) to act as an OpenFlow switch that decides whether each network flow goes across the packet or the optical networks. We extend the OpenFlow protocol in OVS to tag packets with optical circuits. These tags will later determine the time when packets should transit through the emulated OCS. This allows us to define forwarding rules using the OpenFlow protocol. If virtual machines (VMs) are used, their packets are delivered to OVS through a virtual NIC (vNIC), which can forward packets back to other vNICs, to an EPS or the emulated OCS, allowing the experimentation of cloud-based virtualized workloads. Note that since both networks are connected to the same software switch, it is also possible to explore hybrid networks composed by EPS and OCS data planes [4], [5]. However, we leave the exploration of such topic for future work and focus on the challenges to provide an accurate emulation of the optical data plane.

After software switching, we add a kernel module with a new Linux queueing discipline (Qdisc) before the network. For data routed towards the emulated OCS (Optical NIC in Fig. 2), our Qdisc enqueues packets into virtual output queues (VOQ), each distinguished by a circuit destination. Meanwhile, the module also listens for synchronization messages from the OCS Controller on the control network, which announces the availability of new circuits in the emulated OCS data plane. These messages are delivered by a sniffer that we insert into the NIC driver for quick, low latency reaction. If a circuit with packets stored in its corresponding VOQ becomes available, the Qdisc dequeues these packets subject to a token scheme (Sec. IV-B) that controls how packets should be transmitted along the circuit. For the Packet NIC (Fig. 2) connected to the Packet Network (EPS), we use a simple FIFO queue.

B. Precise control of packet transmission

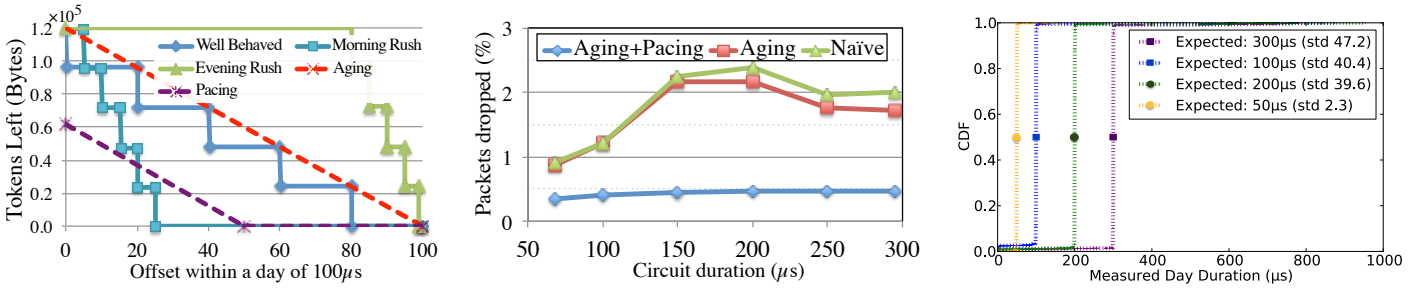
Circuits are emulated with fine-grain control over NIC data transmission through distributed coordination of leaky token buckets. To ensure precise time control of the packets leaving our servers, we disable TCP Segmentation Offloading (TSO). With TSO, the OS relies on the NIC to segment TCP packets according to the network MTU, with the NIC sending data when the segmentation is complete. This removes our ability to decide how many bytes are sent on each circuit; i.e. to decide the precise time to stop feeding data to the optical pipe because it is being teared down. We show associated CPU costs for disabling TSO in Sec. V-A.

For each day, we start a token bucket that controls how many packets can be transmitted from a VOQ. A token in the bucket correspond to a byte transmitted. A naive token scheme might just restrict the transmission of $D * R$ bytes across a circuit, where D is the duration of the day and R is the link rate of the NIC in bytes/s.

One problem with this scheme occurs when a large group of packets become ready for transmission at the end of the day, a situation we call *evening rush*. This can cause the NIC buffer to hold more packets than we can transmit in the remainder of the day. This situation happens because the OS hands off packets to the NIC much faster than the NIC can transmit them. Figure 3a shows this scenario for days of 100 μs and a connection that supports a link rate of 10 Gb/s (125000 bytes may be transmitted in 100 μs). We avoid this by *aging* tokens throughout the day, such that for a day with δ secs remaining and link rate of R bps, only $\delta * R$ bytes can be transmitted.

The less intuitive problem occurs when a large group of packets arrive at the VOQs at the beginning of the day, a scenario we call *morning rush*. This can trigger change of behavior in NICs, and cause extra delays in packet transmission. Even with TSO disabled, packet transmission in most high performance NICs is still done asynchronously, involving both local NIC memory and main memory. When NIC TX queues are full, it can postpone main memory access to avoid packet drops and/or increase performance [19], [20], delaying packet transmission (for example, batched TX at high load can reduce Idle/Active transitions, saving energy). In our servers, if more than 80 KB of data was handed to the NIC at once, packets start to get delivered to the NIC with up to 300μs of delay. If that happens, it is more likely that the NIC decides when to send packets and we lose control over TX times. To solve this, we use a *pacer*: we track the difference between the number of *aged* tokens and the total tokens left for the VOQ to transmit packets. If the difference grows above 80 KB, the dequeue function that hands packets to the NIC reschedules itself to try again in a future software interruption. We call this scheme *pacing* tokens. In the future, NICs with improved latency might also be used as alternative to this problem [20].

Figure 3a summarizes TX synchronization problems and the benefits of improved token management. We also characterize the improvement that the different token schemes have on packet loss rate with an all-to-all UDP traffic generator between 23 servers. If a packet is sent when a day is over, or if it cannot get through the circuit before the day is over, it is dropped (we discuss synchronization for both sides of the links in Sec. IV-C). Figure 3b shows the average loss rate



(a) Effect of enhanced token management

(b) Impact of token mgmt. on packet loss

(c) Day lengths measured at servers.

Fig. 3: Performance of our token scheme (a and b), and validation of our software synchronization module (c).

per server as a function of day duration ranging from 61 to 300 μs for three token schemes. The naive scheme loses an average of 1.79% packets. The *aging* token scheme performs better, losing 1.65% packets on average. The token scheme that employs both *aging* and *pacing* loses only 0.44% packets on average, 4× less than the naive scheme. Interestingly, the loss rate goes down as day length increases from 200 to 250 μs for both the naive and *aging* scheme. This occurs because the day duration begins to start matching the increase in delay from the NIC batch mode.

C. Synchronization Controller

A major challenge in emulating a μs OCS is that data transmission across all OCS ports in a day needs to be synchronized with the current circuit set (see Sec. III). In Mordia [1], a field-programmable gate array (FPGA) was connected to 1) all end-nodes using the OCS and 2) all OCS's Wavelength Selective Switches (WSS), which perform the actual optical switching to establish circuits. The FPGA delivers commands that change WSSs and notifies servers, allowing TX (servers) and RX (WSSs) to match the current active circuit set simultaneously.

We evaluate two approaches to accomplish the same task using an EPS control network. Our first attempt uses a NetFPGA to send synchronization messages to our servers at configurable time intervals with the FPGA's hardware precision. These time intervals correspond to OCSEMU's parameters, day and night. In the second attempt we use only software: we dedicate one server to perform the synchronization, with one of its CPUs spinning in a busy loop announcing circuit setup and tear down events at specific times, according to given day/night, through the EPS. This allows greater flexibility in circuit scheduling, as it can be changed using a software defined approach. The circuit announcement messages contain only MAC addresses (required for EPS forwarding) and circuit information and are not processed by the TCP/IP stack.

Our software approach is implemented in a kernel thread that expects values for *day* and *night* parameters. At the start of a day, the kernel thread pre-computes broadcast frames with circuit information for the next day, and then busy waits checking the CPU time stamp counter (*rdtsc*). When it is time to change the OCS topology, the kernel thread sends the frame to the packet switch, which broadcasts it to each server in the cluster. To avoid OS scheduler interruptions, we use CPU core isolation. We bind the kernel thread to a core that was previously isolated from the OS scheduler. We found no significant difference in terms of precision for the hardware (NetFPGA) and software control, as the major source of error comes from the delay to receive synchronization frames.

V. Evaluation

We focus our evaluation on the CPU load of our emulator and its fidelity to results achieved with an actual μs OCS. We start analyzing the CPU cost of keeping the network TX control in software. We then evaluate OCS emulation behavior. Our testbed is composed of 23 HP DL380G6 servers with Myricom 10G-PCIE-8B2-2S NICs and Cisco 5596UP EPS[2].

A. CPU Overhead

We conduct a study of NIC features and their impact on overall network throughput and CPU overhead. Table I lists all features that directly affect the performance of the system. They are checksum offloading for TX and RX, TSO, scatter gather I/O (to allow non-contiguous Direct Memory Access (DMA) reducing the need for memory copies), and NIC coalescing (which reduces the number of times the processor is interrupted by the NIC, restricting network packet processing to a few 10's μs intervals). For CPU load, we list the OS time (*%sys*) and software interrupt (bottom-half) time (*%sirq*). The former is spent in the network stack code, mostly responsible for TCP/IP processing. The later corresponds to low-level networking code (i.e. for software switching, queuing, etc), that takes place after network stack. It is important to note that *sys* and *sirq* code can run on different CPU cores, but we restrict our workloads to one core in this experiment, to report CPU load as % of a single core. Results are the average of 20 readings taken every second at the sender.

In this experiment, we use two nodes of our testbed as sender and receiver. We generate TCP traffic using the micro-benchmark *netperf*, and gradually turn off NIC features while observing throughput and CPU load. With all NIC features turned on, we notice that the CPU is able to saturate the capacity of our NICs, with 9.89Gbps of traffic reported by *netperf*. Disabling NIC coalescing increases the CPU load of the OS, as software interrupts run more frequently due to non-batched interruptions from the NIC. Turning off TSO causes a slight increase in CPU load, indicating that most of the load from coalescing is from the OS scheduler itself, and not from the network stack.

After turning scatter gather I/O off, we notice that the CPU load, as well as throughput, decreases. This indicates that the bottleneck moved from the NIC to data copy. Further reductions in throughput, and increase in CPU load, take place when we disable checksum offloading, as the OS is now responsible for computing them. Disabling TX checksum has higher impact, as the RX side processes only TCP ACKs.

[2]Note that EPS latency restricts *night* lengths. Our EPS offers 10μs port-to-port latency, but others can reach sub-μs latency, as the Arista 7124FX.

TABLE I: Impact of NIC Offloading on Throughput and CPU

Checksum TX	*Checksum* RX	SG IO	TCP Seg.	NIC Coal.	% CPU sys	% CPU sirq	Thput *(Gbps)*
ON	ON	ON	ON	ON	41.9	7.4	9.89
ON	ON	ON	ON	OFF	63.7	28	9.89
ON	ON	ON	OFF	OFF	63.8	28.9	9.89
ON	ON	OFF	OFF	OFF	38.7	11.4	8.03
ON	OFF	OFF	OFF	OFF	43.2	12	7.98
OFF	OFF	OFF	OFF	OFF	41.8	8.6	6.77

These results show that the CPU is able to saturate the NIC with checksum and scatter gather turned on, so we leave those features enabled during the OCS emulation. Also note that software interrupt time, where OCSEMU is implemented, is small compared to system time. We reserve one out of 16 cores for software interrupt processing, and other system code is allowed to use the other 15 cores. Under higher link speeds (40Gbps) and servers with extra cores, we can use parallel network processing and different architectures to scale OCSEMU, such as in IsoStack [21]. Note that the token scheme and packet scheduling at the host is simple enough to be implemented in a NetFPGA NIC. Thus, in future 100Gbps NIC servers, the server side of OCSEMU can use a hardware implementation for 100Gbps OCS emulation.

B. Validation and Application Performance

This section validates the distributed OCS in two ways. First, we analyze the software synchronization mechanism using fine-grain measurements provided by our NIC. Next, we compare the throughput offered by the emulated OCS with the results from Mordia [1].

1) Synchronization accuracy: We evaluate the precision of our synchronization controller (Sec. IV-C) using our NIC's API, which allows us to capture packet timestamps with precision of ± 500 ns. We test the controller with various combinations of day and night lengths, and use the timestamps of the packets received by our NIC to calculate the perceived length of days (i.e. active circuits, or intervals when nodes are allowed to send traffic) and nights at the servers. At first, we measured standard deviations of days and nights in range of 0.2μs to 48μs, indicating some noise in the synchronization. Concerned that the controller's kernel thread was experiencing interference, we ran the same experiment with our NetFPGA solution (Sec. IV-C), and found the very similar results.

To understand the problem better, we turn our attention to a CDF of measured day durations, in Figure 3c (CDF of nights is similar, thus omitted). From the CDF, we see that the majority of days (>98%) respect the expected day duration configured at the controller. At the extremes of the graph, we could see a few days of 1000μs, what indicates jitter in 1% of our measurements. When a day lasts longer than it should, consecutive days are also affected, which in turn last for 0μs. However, this happens less frequently with shorter days/nights, given the smaller measured deviations.

We verify that the OS jitter in servicing hardware interrupt is comparable to our measurements. This can be done by scheduling the systems high precision clock to trigger an interrupt at a specific time in the future and check the time that the interrupt service routine starts executing. The results are less than 10μs on average, even under high server load. Thus, the reason for more errors with longer days/nights is the long (less frequent) signaling interval, which results in higher likelihood of OS-induced delays on software interrupt execution. This causes some days to appear longer than they actually are. However, the aging token scheme prevents OCSEMU from sending extra packets on longer days, and as a result we should see minor deviations from the actual amount of packets transmitted in a day in $\sim$1% of the cases.

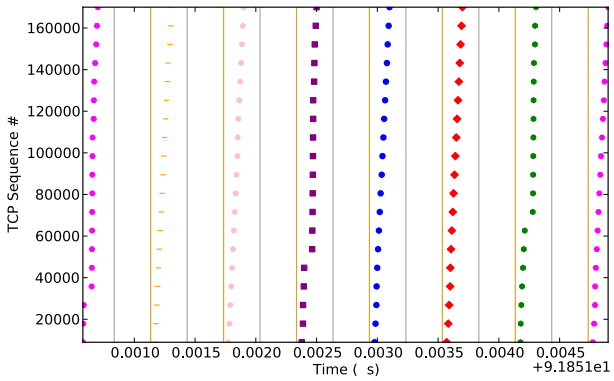


Fig. 4: All-to-all RX over time seen from one node. Colored dots are packets with TCP Sequence number in the Y axis. Each color represents one sender.

2) Throughput validation with all-to-all traffic: The second phase of our validation verifies that OCSEMU is able to deliver similar all-to-all throughput as an actual OCS prototype (Mordia) [1]. We decided to compare with such OCS because, to date, this is the only μs OCS that reported results with server-generated traffic using TCP. As mentioned in Sec. III, links to end-nodes can be scheduled by cycling through all possible circuits in a round robin fashion. We use this scheduling scheme, as in Mordia, and generate the same all-to-all traffic pattern using the microbenchmark *netperf*.

Figure 4 exemplifies OCS behavior under all-to-all traffic. Colored dots have their X axis value based on the timestamps of a RX packet capture from one of the nodes. Values in Y axis represent (relative) TCP sequence numbers from captured packets, and each distinct color represents a different source IP (sender). In this case, OCSEMU is configured with *day*=*night*=300μs, with vertical lines representing beginning of days (golden color) and nights (gray). The figure shows that OCSEMU can schedule data transmission according to emulated circuits enforced by the controller.

Figure 5 combines TCP throughput achieved on an EPS, Mordia, and OCSEMU. The experiment uses fixed nights of 35 μs and variable days from 61–250 μs. As in the Mordia evaluation, the all-to-all TCP throughput achieved in our EPS was 6.4 Gbps/server, so we did not expect our OCS emulator to go beyond that. However, we notice a major difference of $2Gbps$ between our emulator and Mordia. We hence introduced logic in our kernel module to disregard night synchronization messages and strictly discard packets that were partially or entirely received during nights, to measure the impact of drops on throughput. This explores the interaction between synchronization controller imprecisions, discussed in the previous section, and packet drops. The result labeled *"OCS Emu. (Loss=X)"* shows the throughput with this feature, where X indicates the allowance of drops into the night. This indicates that Mordia could provide much better TCP throughput if the 0.5% drop rate published in the paper was avoided. As with Mordia, the imprecise synchronization of 1% in our emulation causes packet drops. The impact is significant because TCP

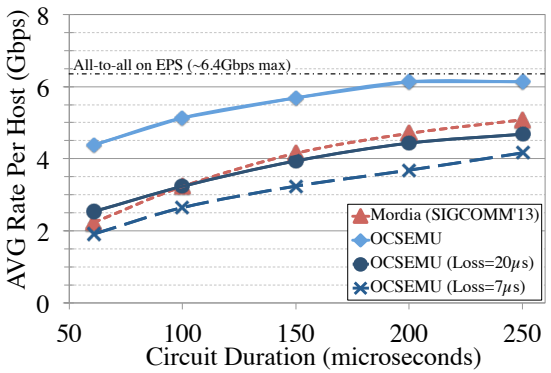


Fig. 5: OCSEMU and real OCS throughput for all-to-all TCP

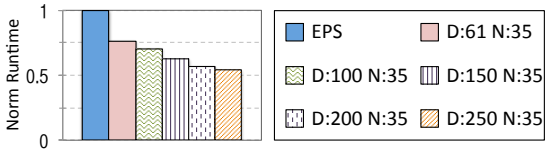


Fig. 6: NASA Parallel Benchmarks (BT) [22] on OCSEMU

congestion control "backs-off" multiplicatively, and at high speed this results in severe reduction of TCP throughput.

3) Application Performance: To evaluate the impact of fast OCS scheduling on applications, we deployed an unmodified distributed application from the NASA Parallel Benchmarks [22] in our testbed. The application is a solver for systems of linear equations with tridiagonal matrixes. Nodes compute parts of the problem and use Message Passing Interface (MPI) to communicate partial solutions. Figure 6 shows the execution time of the application using OCSEMU with different day (D) durations (in μs) normalized to the (optimal) runtime of an EPS interconnect of same bandwidth. The average runtime of the application on the EPS is 256.07s.

As expected, faster OCS provides reduced latencies and improves the execution time of the application. However, we also note that OCS availability has a bigger impact than duty cycle, resulting in significant increase in application runtime (50% to 100%, depending on day/nights) compared to the runtime when an EPS is used. This happens because the solver uses an iterative algorithm with several steps, and the extra latency induced by the round-robin scheduler impacts the execution time of each step, increasing the overall runtime of the application. This experiment demonstrates two points. First, OCSEMU can be used to experiment unmodified applications in an emulated OCS interconnect providing detailed access to profiling and performance analysis. Second, OCSEMU is extensible, and can be used to experiment new OCS controllers that improve performance of applications with different demands and network patterns. We are currently experimenting performance of other applications on OCSEMU. However, we leave the exploration of this topic for future work.

VI. Conclusion

Compared to Electronic Packet Switches (EPS), Optical Circuit Switching (OCS) offers scalable bandwidth and high energy efficiency, but the technology pays the price of decreased flexibility. We described and evaluated OCSEMU, an OCS Emulator designed to contribute to the research progress in application development and performance analysis of these emerging fast circuit switched environments. Our results show that OCSEMU can match the performance and behavior of a real μs OCS, allowing many research groups to contribute to this area without having to build a state-of-the-art OCS.

Acknowledgements

This work was partially supported by the CIAN NSF ERC under grant #EEC-812072 and EU ACROSS program.

References

[1] G. Porter, R. Strong, N. Farrington, A. Forencich, P. Chen-Sun, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat, "Integrating Microsecond Circuit Switching into the Data Center," in *SIGCOMM*, 2013.

[2] "Huawei CloudEngine 12800 Series Switches Datasheet."

[3] K. Chen, A. Singlay, A. Singhz, K. Ramachandranz, L. Xuz, Y. Zhangz, X. Wen, and Y. Chen, "OSA: An Optical Switching Architecture for Data Center Networks with Unprecedented Flexibility," in *NSDI*, 2012.

[4] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat, "Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers," in *ACM SIGCOMM*, 2010.

[5] H. Liu, F. Lu, A. Forencich, R. Kapoor, M. Tewari, G. M. Voelker, G. Papen, A. C. Snoeren, and G. Porter, "Circuit switching under the radar with reactor," in *NSDI*, 2014.

[6] N. Farrington, A. Forencich, G. Porter, P.-C. Sun, J. E. Ford, Y. Fainman, G. Papen, and A. Vahdat, "A multiport microsecond optical circuit switch for data center networking," *Photonics Technology Letters, IEEE*, 2013.

[7] S. Han, T. J. Seok, N. Quack, B.-W. Yoo, and M. C. Wu, "Monolithic 50x50 mems silicon photonic switches with microsecond response time," in *Optical Fiber Communication Conference (OFC)*. OSA, 2014.

[8] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *ACM SIGCOMM*, 2011.

[9] E. Weingärtner, F. Schmidt, H. Vom Lehn, T. Heer, and K. Wehrle, "Slicetime: a platform for scalable and accurate network emulation," in *NSDI*. USENIX, 2011.

[10] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. S. E. Ng, M. Kozuch, and M. Ryan, "c-Through: Part-Time Optics in Data Centers," *ACM SIGCOMM*, 2010.

[11] R. Kapoor, A. C. Snoeren, G. M. Voelker, and G. Porter, "A study of nic burst behavior at microsecond timescales," in *ACM CoNEXT*, 2013.

[12] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *ACM CoNEXT*, 2012.

[13] M. Carson and D. Santay, "Nist net: a linux-based network emulation tool," *ACM SIGCOMM Computer Communications Review*, 2003.

[14] L. Nussbaum and O. Richard, "A comparative study of network link emulators," in *Simulation Multiconference*. SCS, 2009.

[15] S. Azodolmolky, M. N. Petersen, A. Manolova Fagertun, P. Wieder, S. R. Ruepp, and R. Yahyapour, "Sonep: A software-defined optical network emulation platform," in *Optical Networks Design and Modeling Conference*. IEEE, 2014.

[16] D. Gupta, K. V. Vishwanath, M. McNett, A. Vahdat, K. Yocum, A. Snoeren, and G. M. Voelker, "Diecast: Testing distributed systems with an accurate scale model," *ACM Trans. Comput. Syst.*, 2011.

[17] D. Huang, K. Yocum, and A. Snoeren, "High-fidelity switch models for software-defined network emulation," in *ACM HotSDN*, 2013.

[18] N. Farrington, G. Porter, Y. Fainman, G. Papen, and A. Vahdat, "Hunting mice with microsecond circuit switches," in *ACM HotNets*, 2012.

[19] "Fine-tuning your network drivers," http://www.qnx.com/developers/docs/6.4.1/neutrino/technotes/finetune_net.html.

[20] M. Flajslik and M. Rosenblum, "Network interface design for low latency request-response protocols," in *USENIX ATC*, 2013.

[21] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda, "Isostack: highly efficient network processing on dedicated cores," in *USENIX ATC*, 2010.

[22] "Nasa Parallel Benchmarks Website," http://www.nas.nasa.gov/publications/npb.html.