OrbWeaver: Using IDLE Cycles in Programmable Networks for Opportunistic Coordination

Liangcheng Yu
University of Pennsylvania
leoyu@seas.upenn.edu

John Sonchack
Princeton University
jsonch@princeton.edu

Vincent Liu
University of Pennsylvania
liuv@seas.upenn.edu

Abstract

Network architects are frequently presented with a tradeoff: either (a) introduce a new or improved control-/management-plane application that boosts overall performance, or (b) use the bandwidth it would have occupied to deliver user traffic.

In this paper, we present OrbWeaver, a framework that can exploit unused network bandwidth for in-network coordination. Using real hardware, we demonstrate that OrbWeaver can harvest this bandwidth (1) with little-to-no impact on the bandwidth/latency of user packets and (2) while providing guarantees on the interarrival time of the injected traffic. Through an exploration of three example use cases, we show that this opportunistic coordination abstraction is sufficient to approximate recently proposed systems without any of their associated bandwidth overheads.

1 Introduction

The purpose of a computer network is to transmit messages to and from connected devices. The bulk of these messages are sent between two or more end hosts and are intended for use in applications therein (video streaming, web browsing, ssh terminals, stock trackers, etc). It is important to note, however, that networks are also frequently used for other purposes that are not directly related to end-to-end application traffic. These uses include but are not limited to control messages, keepalives, and probes.

In some cases, this second category of messages is sent over dedicated networks (e.g., an out-of-band control plane). Nevertheless, a significant portion is not, and for good reason. Multiplexing the traffic over a unified network results in more efficient resource utilization and helpful fate-sharing properties. For many uses, it is also required for correctness. For instance, active probing generally relies on the probe facing the same network conditions as normal traffic.

For in-band coordination, there is often a choice between fidelity and overhead. More so as many protocols use high-priority messages that directly cut into network capacity. For example, when deciding on an appropriate interval for sending routing-protocol keepalive messages, sending keepalives more frequently results in faster failure detection but at the cost of many extra packets in the network. Similarly, while techniques like congestion tagging [3, 22] and in-band network telemetry [27] can provide timely information about the recent state of network paths, they require either extra probe packets or space in the headers of existing packets, both of which occupy valuable bandwidth.

Given this tradeoff between fidelity and overhead, today's networks end up settling for a little bit of both. In some cases, the sacrifices are modest; in others, network operators are forced to limit the aggressiveness of their systems despite evidence of the benefits of finer granularity [6, 49]. In this paper, we argue that for a diverse set of protocols, the sacrifice is entirely unnecessary—systems can coordinate at high-fidelity with a near-zero cost to usable bandwidth and latency. In short: we can have our cake and eat it too.

Our system, OrbWeaver, is a framework for the opportunistic transmission of data across today's programmable networks. OrbWeaver takes advantage of gaps between user traffic and 'weaves' (i.e., injects) into *every* such gap customizable IDLE packets that convey information across devices. For modern, high-speed networks, these opportunities are plentiful. Crucially, OrbWeaver provides guarantees about the 'weaved' stream—guarantees on the maximum time between any two packets and guarantees on the impact of the injected packets on user traffic, switch resources, and power draw. A consequence of this predictability is that, even when there is no opportunity to send, the absence of IDLE packets reveals concrete information about the state of the network.

We note that a similar abstraction already exists at the data-link layer. In particular, in today's full-duplex Ethernet standards, the Physical Coding Sublayer (PCS) will fill any gaps in transmission with IDLE symbols [32, 41]. The continuous stream of incoming signals allows receivers to—with no impact to user traffic—test for corruption and link integrity at a fine granularity, even when there is no traffic on the network. Further, by continuing to transmit IDLE symbols after a link integrity issue has been raised, switches can also determine when the link becomes usable again.

OrbWeaver extends this technique to higher layers of the network stack by exploiting the data plane programmability, architecture, and packet generation capabilities of emerging programmable switching platforms. The resulting stream of packets can be used to generalize Ethernet's robust failure detection properties to a broader class of faults; however, its benefits go far beyond L3 failure detection. Rather, we demonstrate in this paper that with proper application, the nearly free communication that IDLE packets provide can be used to eliminate the fidelity-utilization tradeoff of solutions

to several classic problems in networking including clock synchronization and load balancing.

Implementing OrbWeaver's packet weaving presented several technical challenges. First, while IDLE symbols are part of the Ethernet standard and enjoy direct hardware/protocol support, to utilize today's devices and maintain their current performance, OrbWeaver must provide similar behavior without changes to switch architectures. Second, while many systems can benefit directly from opportunistic data transmission, many must continue to operate during periods where user traffic is already occupying all available bandwidth. To address the first challenge, OrbWeaver introduces a co-design of the selective data-plane filtering mechanisms and the rich priority configurations found in modern switches to guarantee minimal impact on user traffic. We verify the approach through a detailed examination of the specifications of the queuing subsystems on a Tofino switch along with experiments that stress-test worst-case behavior. To address the second, we introduce novel mechanisms that exploit IDLE packets and the guarantees of weaved streams to eliminate the bandwidth overheads of existing network protocols. We demonstrate these mechanisms through three case studies.

Our implementation[1] and evaluation demonstrate the efficiency and efficacy of OrbWeaver using real hardware, optical attenuators, and power meters. We find that, despite the introduction of the IDLE stream, OrbWeaver incurs negligible impact on user traffic, the computational/state resources of participating switches, or their power draw. We further demonstrate that this messaging substrate can be used to (re-)design recently proposed systems to eliminate their bandwidth overheads while closely approximating their performance.

2 Motivating Weaved Streams

This section presents the definition of a 'weaved' stream, its motive, and where data plane programmability can help.

Definition. A *weaved stream* is a union of user and IDLE packets traversing a link between two arbitrary network devices that provides two guarantees:

R1 That no link stays unutilized for too long. More precisely, there is some period τ where the interval between any two consecutive packets, d, satisfies $d \leq \tau$.

R2 That the injected packets do not decrease the effective throughput of user traffic or increase their loss rate.

Why weaved streams? Network protocols are, fundamentally, distributed computations that require coordination between different devices—sometimes adjacent devices, sometimes remote devices—for monitoring, control, and management. A perennial problem is how much bandwidth to allocate to these protocols, as each byte devoted to coordination is a byte that could have been used for user traffic instead. This tradeoff has tangible effects for many networking tasks:

- *Failure handling:* A common strategy for detecting the failure of remote network devices is the use of continuous keepalive messages. Here, each node periodically sends a keepalive to each of its neighbors; if a neighbor ever stops sending keepalives, nodes assume that they have failed. Fundamentally, the period between keepalives bounds the speed at which we can detect failures. Unfortunately, because keepalives are most accurate when sent over the same or higher-priority channels as user traffic, their sending rate is typically kept low (e.g., at a period of O(100 ms)) at the cost of slower detection.
- *Clock synchronization:* Prior work has also noted the utility of synchronizing network devices [29], e.g., for coordinated network updates [36, 45] or real-time streams [13]. Clock synchronization protocols typically pass messages that periodically compute the drift between the clocks of participating machines. Constant changes to not only the relative clocks but also the relative clock *rates* mean that more frequent updates can provide more accurate synchronization (at the cost of additional packets in the network, typically configured at a high priority).
- *Congestion notification:* Finally, this tradeoff can be seen in the detection/communication of congestion and current load. ACKs (and their corresponding loss/RTTs) are a particularly common method for inferring the presence of congestion, e.g., in TCP NewReno. As others have noted [3, 26], however, there are also advantages to more explicit signaling of the current congestion and queue statistics. Unfortunately, while effective, these statistics typically occupy packet header space or introduce additional packets into the network.

Over the years, network architects have developed many workarounds. These include hardware changes [29, 32], co-opting unused fields in headers [3, 50], carefully balancing the tradeoff for a particular service-level expectation [7], or otherwise coming to terms with the cost of coordination. Outside factors can guide the above decisions, such as whether ACKs are already necessary (e.g., for reliability) or if extraneous fields can be eliminated. However, in this paper, we ask a more fundamental question: are these tradeoffs necessary?

To that end, OrbWeaver is a framework for implementing network coordination that does not interfere with user traffic. OrbWeaver's weaved streams are both opportunistic and highly predictable—consuming every inter-packet gap of sufficient size but no more. Not every protocol can be implemented solely using weaved streams (though many can benefit from it). Even so, we demonstrate that at least for the three use cases above, weaved streams are sufficient to approximate state-of-the-art systems while reducing their impact on user traffic to virtually zero.

Why are there gaps? Usable gaps between packets can occur for many reasons, the most basic being application-level patterns and TCP effects. Indeed, prior work [37, 49] and

[1] Code is available at `https://github.com/eniac/OrbWeaver`

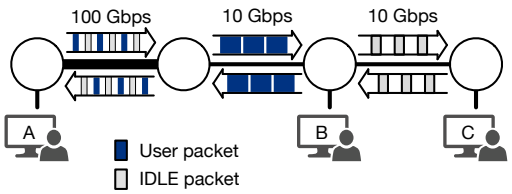


Figure 1: An example OrbWeaver-enabled network with four switches and three end hosts (connected with 10 Gbps links). A single two-sided connection $A \leftrightarrow B$ occupies the network, but a significant portion remains unused. Gaps between packets can occur for many reasons, but OrbWeaver can weave IDLE packets into all of those gaps.

our conversations with several large clouds/ISPs verify that micro-/milli-second inter-packet gaps are ubiquitous, even in networks that primarily handle large bulk-data transfers.

Gaps can also happen for structural reasons. For example, consider Figure 1 (sans IDLE packets). In it, a single connection $A \leftrightarrow B$ occupies all usable end-to-end bandwidth. Even if A and B pace packets perfectly, no host can send additional packets without displacing the existing user traffic, despite significant opportunities to do so (because of, e.g., congestion, link speed changes, and asymmetric connections). These gaps present a chance for opportunistic coordination.

Why now? OrbWeaver's ability to weave IDLE packets into gaps between user traffic is enabled by several features in modern switches: programmable data plane behavior, the capacity for local packet generation, and the ability to fully configure the queuing/prioritization of different traffic classes. We note that none of these are sufficient on their own.

For example, consider strict packet prioritization, which has been used for opportunistic bandwidth allocation [21, 24]. In SWAN [21], for instance, end hosts send low-priority background traffic to capture any bandwidth remaining after handling interactive and elastic services. A naïve application of these techniques, however, is a poor fit for in-network coordination, which occurs between devices in the network (as opposed to end hosts) and typically involves small data sizes that benefit from even short sending opportunities. Figure 1, for example, would not benefit from end-host actions.

3 Generating a Weaved Stream

Before we delve into the potential uses of weaved streams in Section 4, we first detail how to implement the abstraction in today's programmable switches.

Switch model. For simplicity, we primarily focus on the popular Tofino family of programmable networking devices (and discuss generalization to other types of devices in Appendix B). Figure 2 shows a conceptual diagram of the relevant components of the switches we consider. At a high level, when a packet enters from one of the Ethernet ports, its header is extracted by the programmable parser responsible

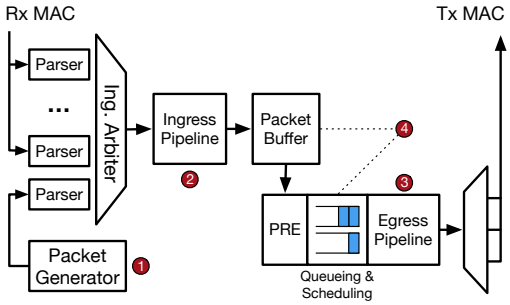


Figure 2: Conceptual diagram of the relevant components of an RMT switch, derived from the switch specifications in [12]. Only a single ingress/egress pipeline are shown. Circled numbers indicate steps and potential points of contention with user traffic that are handled in Section 3.1.

for that port. An ingress pipeline arbiter is then responsible for selecting one of the parsed packets and passing it through the ingress match-action pipeline.

After ingress processing, the packet will be placed in a shared packet buffer until it is ready to be sent out. Instead, the switch uses a shorter 'packet descriptor' for the next steps: optional replication by a Packet Replication Engine (PRE) (e.g., for multicast) and placement onto a per-port egress queue for eventual processing/deparsing. The data plane program and the traffic manager configuration decide whether an incoming packet should be buffered and whether a buffered packet should be enqueued for transmission.

Goal. R1 of the weaved stream abstraction requires a constant stream of packets on every link such that the union of user and IDLE packets satisfies $d \leq \tau$. We note that the optimal guarantee for τ is dependent on both the bandwidth, B, of each link and the MTU of the network. To see why, consider the extreme case where a user is occupying all of the bandwidth of a port i with MTU-sized packets. The receiver on the other side of the link will receive packets at a period of $\tau_i = \frac{\text{MTU}}{B_i}$, with OrbWeaver unable to inject any additional packets without impacting user traffic. Therefore, unless otherwise noted, OrbWeaver uses $\tau_i = \frac{\text{MTU}}{B_i}$ even if smaller IDLE packets would allow for faster injection.

In the worst case when there were zero user packets and N egress ports, the resulting target IDLE-injection rate is:

$$T = \sum_{i=1}^{N} \frac{B_i}{\text{MTU}}$$

For reference, for a 32 port switch with $B = 100$ Gbps and MTU $= 1500$ B, the per-port inter-packet gap, τ_i, is 120 ns, which results in $T = 266.7$M packets/sec.

Constraints. Complicating the injection of IDLE packets into the network are R2 and hardware constraints on the throughput of each switch pipeline, defined in terms of both byte-level bandwidth ($N \times B$) and packet-level bandwidth (proportional to the clock rate of the pipelines). For the latter, switches

typically provide guarantees up to a certain minimum packet size, and best-effort behavior for very small packets.

3.1 Mechanism Overview

OrbWeaver's IDLE-packet weaving leverages a combination of features found on our target platform: data-plane packet generation, data plane programmability, and fine-grained arbiter/scheduler configuration options. The switches' onboard per-pipeline packet generator modules, in particular, form a convenient substrate for our techniques. Using these modules, a network operator can create packets with predetermined content at a predetermined rate.

In principle, one could configure the generators to create packets at a rate T (thus providing OrbWeaver with its consistent stream of packets to convert into IDLE packets). Unfortunately, in practice, these generators do not have nearly enough capacity to satisfy the requirements of OrbWeaver. Moreover, blind injection of packets may interfere with the throughput, latency, or loss of user traffic. Instead, OrbWeaver uses the selective amplification method described below.

❶ **Packet generation.** The IDLE stream generation of OrbWeaver begins with a low-rate but predictable stream of generated IDLE packets. The focus of this process is to provide a 'seed' stream with an emphasis on regularity; amplification up to T occurs later in the pipeline. More specifically, the generator module is configured to send a packet every $\frac{\tau_{\min}}{2}$ secs, where $\tau_{\min}$ is the minimum τ_i of any port on the pipeline.

There are two important aspects of this seed stream. The first is that the rate is double that of $\tau_{\min}$ in order to provide a degree of oversampling for the subsequent optimizations without sacrificing guarantees on the eventual spacing of packets. The second is that the IDLE packets are configured with a strict high priority at the ingress arbiter so that the packet will always be serviced as soon as it is generated. While this implies that IDLE packets are preferred over user traffic in the ingress pipeline, the low rate of this seed stream means that OrbWeaver incurs $<$1.5% overhead even for the worst case of minimum-sized packets sent at $\tau_{100\,\text{Gbps}}$ (denoting the optimal τ_i for a 100 Gbps link). More typical packet sizes and utilization eliminate the overhead.

❷ **Amplifying the stream on-demand.** OrbWeaver takes the low-rate seed stream above and amplifies it, potentially up to the full rate T, by leveraging another hardware feature found in modern switches: flexible multicast. In Figure 2, this behavior is implemented in the PRE, which can replicate a packet descriptor to the egress queues at line rate.

Unfortunately, the naïve approach of replicating a packet to every egress queue every $\tau_{\min}$ seconds can crowd out normal multicast packets and waste significant egress capacity. More specifically, there are two instances where it is not necessary to multicast a packet to a particular port i:

1. If the port is slower than the maximum speed, then sending at $\tau_{\min}$ will be too fast by a factor of $\frac{B_{\max}}{B_i}$.
2. If a user packet was already sent to the egress port recently, sending an IDLE packet is unnecessary.

OrbWeaver addresses both cases by oversampling the sending history of each port (at rate $\frac{\tau_{\min}}{2}$) and then selectively filtering/multicasting toward only the ports that need an IDLE packet. When a port is has bandwidth $B_i < B_{\max}$, the switch downsamples the IDLE packets by configuring two multicast groups (one with port i and one without) and picking the one with i every $\lceil \frac{B_{\max}}{B_i} \rceil$ packets. Similarly, if a port has sent a packet (user or IDLE) in the past $\frac{\tau_{\min}}{2}$ seconds, we can select a multicast group that does not contain the given port.

Concretely, this filtering step uses a single stateful register entry with a bit width equal to the number of ports attached to the pipeline. In essence, the register is a bitvector where each bit represents whether we have sent a packet to the associated port within the last $\frac{\tau_i}{2}$ seconds. For every incoming seed packet, if the associated bit is 1, we omit the port and flip the bit to 0; if the bit is originally 0, include the port in the multicast and flip the bit to 1. Specifically:

```
user packet: filter_reg |= 1 << egress_port
seed packet: filter_reg ^= speed_mask
```

When all ports are the same speed, `speed_mask` is always $2^N - 1$; for hybrid configurations, the ith bit is 1 for every $\lceil \frac{B_{\max}}{B_i} \rceil$ packets and 0 otherwise. After updating the register, OrbWeaver multicasts the current seed packet to the multicast group specified by `filter_reg` (in particular, its value before the xor)—if and only if bit i in the multicast group ID is 0, port i is included in the multicast.

In principle, a direct application of the above filtering step guarantees that the PRE will have enough bandwidth for all user multicasts, assuming that each user multicast results in at most one packet on each egress port. Two aspects of modern switch design potentially complicate this design.

The first is that today's switches typically cannot support a unique multicast group for each of the 2^N possible combinations of target ports. OrbWeaver addresses this by reducing the number of groups by coalescing ports into groups of M such that, if any port in the set has its bit in `filter_reg` set, the entire set receives the multicasted packet. This approach trades a factor of 2^M reduction in the number of multicast groups for a worst-case $\frac{M-1}{N}$-factor decrease in PRE bandwidth. The second is that modern switches are often composed of different pipelines, each supporting distinct packet generators, sets of registers, and groups of ports. Lack of visibility across pipelines means that `filter_reg` may only track local sends, which can also lead to higher PRE usage.

We note, however, that in both of the above cases, OrbWeaver will only incur false negatives (and no false positives) of user packet presence, thus satisfying R1. We also note that very few modern networks are continuously multicasting to all ports at near line-rate.

❸ **Weaving the IDLE stream between user packets.** After the stream is amplified, it reaches the egress queues and

pipeline of the switch. To bound the impact of the stream on user traffic, OrbWeaver configures its packets to have a strictly lower priority than any other user traffic on the same port. If there is user traffic to send, the IDLE packets will not impact them; if there is no traffic to send, the IDLE packets will be sent at a minimum rate of τ_i per port i. The only potential impact to the latency/throughput of user traffic is when an IDLE packet is scheduled just before a user packet arrives, in which case the user packet will be delayed by at most $\texttt{pkt\_size}/B_i$. The delay is only incurred once per packet burst, which implies a bound on OrbWeaver's end-to-end impact on latency and throughput.

Upon arriving at the ingress pipeline of the downstream switch, the packets will be dropped. This also has near-zero impact on user traffic as IDLE packets are only received when the upstream switch has nothing to send.

❹ **Managing the packet buffer and egress queues.** Finally, through the above process, there are two primary places where IDLE packets can compete with user packets for memory in addition to bandwidth. The first is the per-egress output queues that hold packet descriptors before they are serviced by the egress pipeline. The second is the shared packet buffer that stores packet contents until they are sent out on the wire.

To bound the impact of OrbWeaver on both resources, we statically carve the buffer using egress and ingress buffer accounting mechanisms, respectively. For the former, we note that the queue for IDLE packets (the lowest priority queue for the port) is distinct from those of user packets. This queue only needs to be one cell deep as another IDLE packet is guaranteed to arrive in a timely fashion, and thus, the impact on aggregate memory capacity is negligible. For the latter, we can likewise keep the required buffer shallow because of the guarantees of the packet generation process. Specifically, we can confine the IDLE packets to a fixed-size, non-shared region of the packet buffer. The buffer only needs to have a depth equal to the sum of the egress, per-port IDLE-packet queues plus a small amount of headroom for any potential cycle-level processing delays. This is $< 0.01\%$ of the total buffer size of a typical modern switch.

3.2 Evaluating the Weaved Stream

In this section, we delve deeper into OrbWeaver's potential impacts on user traffic. We do this with the assistance of a prototype implementation on a $2\times$ Wedge 100BF-32X testbed. Additional experiments can be found in Appendix F.

3.2.1 Can OrbWeaver Inject at Rate T?

To demonstrate that our approach can achieve T on a fully provisioned switch, we validate it empirically. Specifically, we configure a switch with all 32 ports active and running at a full 100 Gbps. We then configured the switch's packet generator module to generate seed packets at a rate of $2/\tau_{100\,\text{Gbps}}$ and then multicast every other IDLE packet to all ports.

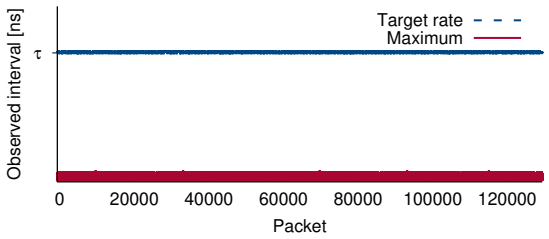


Figure 3: An empirical evaluation of the switch's capacity to generate IDLE packets. Packets were injected to all ports, but the graph depicts the observed inter-packet gap at only one of those ports. Results are shown for both the target rate ($B_i = 100$ Gbps, MTU $= 1500$ B) and the maximum achievable rate. y-axis omitted to protect confidential information.

Figure 3 shows a time series of the interval between IDLE packets, as observed by the egress pipeline of a single port. To record the series, we maintained a ring buffer (implemented via a data plane register) of the difference between the current `egress_global_tstamp` and the previous. The observations were maintained in the egress pipeline and for a single port (other ports' results are identical).

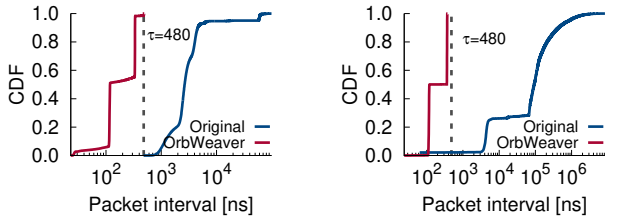
We find that, not only is the injected stream able to achieve $\tau_{100\,\text{Gbps}}$ for every port simultaneously, the observed rate is stable across packets. Further, increasing the amplification factor of the multicast configuration enables IDLE packet generation more than an order of magnitude faster than the target interval, $\tau_{100\,\text{Gbps}}$. Among other implications, this means IDLE packet injection is robust to higher bandwidth and lower MTUs, even without improvements to packet replication capacity.

3.2.2 Can OrbWeaver Bound Packet Gaps?

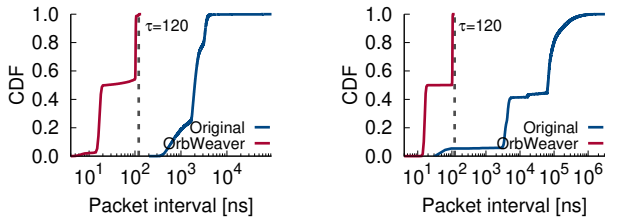
In addition to being able to generate IDLE packets at rate T, R1 also requires regularity in the form of a bound on the gap between packets. We note that Figure 3 already demonstrates the regularity of this gap on a switch without traffic. We also note that in the other extreme (when ports are always congested), R1 is trivially satisfied.

In this section, we extend these results to a network with burstiness and varying levels of traffic. Specifically, we use a hardware testbed consisting of two OrbWeaver-enabled switches (A and B) and a set of servers connected to A. User traffic is passed hosts$\rightarrow A\rightarrow B$ with amplification to fully utilize the ports at B. For this experiment, we used `tcpreplay` and pcap traces from an ISP backbone [9] and a cloud data center [8]. We set up a register in the ingress pipeline of the downstream switch B to record the distribution of the interval between consecutive packets.

Figure 4 shows the results for a single 25/100 Gbps port. Without OrbWeaver, very few intervals are under τ for the target link speed, and the tail is very long. OrbWeaver, on the other hand, is able to weave in IDLE packets to guarantee an upper bound on the packet interval regardless of the original traffic pattern. In particular, for a configured generation interval of t ns, out of 2.14×10^9 interarrival periods, the max-



(a) 25 Gbps ISP backbone (b) 25 Gbps data center



(c) 100 Gbps ISP backbone (d) 100 Gbps data center

Figure 4: Observed intervals between packets with/without OrbWeaver's weaved stream. The dotted line shows the ideal period τ for each link speed. Without OrbWeaver, the maximum interval was >100s of ms but we truncate for readability.

imum observed interval was $(t+3)$ ns (observed for only 32 intervals). The discrepancy is likely due to either clock drift or the aforementioned cycle-level processing delays. Notably, the presence or absence of cross traffic had negligible effect on the frequency of these 3 ns outliers so in practice, we can set $t = \tau - 3$ and achieve reliable results.

Explanation. The regularity of OrbWeaver's weaved stream derives from the architecture of the switch and the mechanisms of OrbWeaver. From the components of Figure 2, the parser used by the packet generator is separate from those of the external traffic, the ingress pipeline grants strictly higher priority to the generated packets over external traffic (user or IDLE), and the packet buffer protects IDLE packets from interference through static reservations for worst-case capacity. When combined, a generated IDLE packet can only be delayed through HoL blocking when an external packet arrives just before the generated packet. For unicast packets, this is a 1-cycle delay; for full broadcasts, this is up to an N-cycle delay (which is short for today's high-speed networks).

At the egress pipeline, the priorities are reversed: IDLE packets are set to a strictly *lower* priority than user traffic. This change stems from a change in objective: in the egress pipeline, it is no longer necessary for the IDLE packets to be sent at a precise rate; instead, the goal is to send *any* packet at above the minimum rate, τ_i. Choosing a user packet instead of an IDLE one can only decrease the inter-packet gap.

Note that, in a Tofino, these priorities (unlike those at the ingress) are only effective within their respective ports. Thus, the switch will send a low-priority packet on port i even if there is a higher-priority packet queued for a different port. As long as the average packet size is above the minimum for line-rate processing, ports can be considered in isolation.

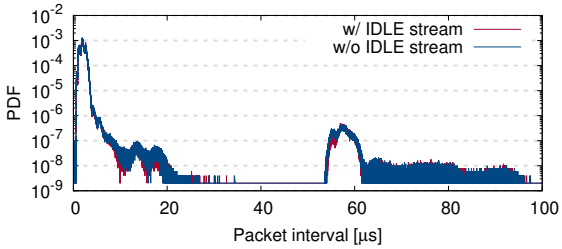


Figure 5: The impact of IDLE packets on user traffic at the ingress pipeline with/without a generation rate of $2/\tau_{100\,\text{Gbps}}$.

3.2.3 Do IDLE Packets Affect External Traffic?

As important as the impact of cross traffic on generated IDLE packets are the impacts of the generated packets on (1) user traffic and (2) incoming IDLE packets. A significant impact on (1) implies violations of R2; on (2), it implies inaccuracy in inter-arrival times and potential violations of R1. We discuss potential impacts in the two pipelines separately.

Ingress pipeline. While OrbWeaver's packet prioritization means that IDLE packets will be preferred over external traffic in the ingress pipeline, its use of multicast amplification reduces their impact to 1.5% of maximum packet-level capacity, with zero impact to byte-level capacity.

To evaluate the practical effects of this overhead, we replayed a real-world packet trace over an ingress pipeline of an OrbWeaver switch. The packet trace was generated using `tcpreplay` and link-level packet traces captured from 10 Gbps Internet routers [9]. To saturate the pipeline, we sped the traces up to match our setup's 100 Gbps per-link bandwidth and replicated them to fill the switch.

We compare two cases. In the first, only the above external traffic is present. In the second, we used the exact same traces but, in parallel, we injected IDLE packets into the same pipeline just as we did in the previous subsection. In both cases, we measured the packet count and interarrival times of user packets in the ingress pipeline with the help of stateful registers that aggregate both statistics.

We find that, for the speeds and packet sizes in the evaluated trace, the throughput and congestion loss of user traffic is the same whether the generated IDLE stream is present or not. The only metric that is impacted is latency, where a slight delay can be introduced each time a generated packet is processed one 'clock cycle' ahead of a user packet; however, this is minor and mitigated by the low frequency of IDLE packet injection. Figure 5 depicts the cumulative impact of this delay using a histogram of the packet interarrival time of the traces, with and without the IDLE stream—the majority of the differences are due to randomness in `tcpreplay` between executions, rather than OrbWeaver.

Egress pipeline. The benefits of the amplification strategy to contention mitigation stop at the PRE, but two other factors take its place in ensuring that user traffic is not impacted in the egress pipeline. The first factor is the filtering step that was introduced in Section 3.1, which prevents superfluous

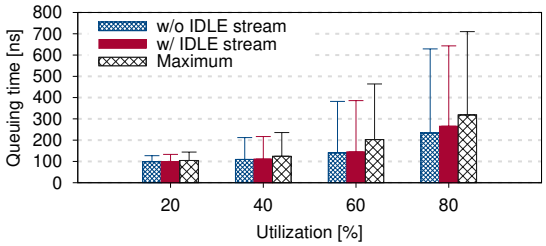


Figure 6: The impact of IDLE packets on the latency of user traffic at the egress pipeline. Results are shown for various levels of average utilization. 0% and 100% are not shown as OrbWeaver becomes trivially optimal. To provide an upper bound on the impact, we disable adaptive ingress filtering and populate the pipeline with only small (64 B) user packets. A real OrbWeaver deployment would have much lower impact.

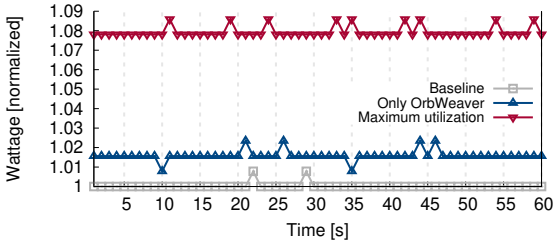


Figure 7: The power draw of a OrbWeaver switch compared to that of an idle (baseline) and a maximally utilized switch. Y-axis is normalized to the average power draw of the baseline.

usage of both the PRE and egress pipeline when the egress ports are already occupied. For IDLE packets that are not filtered in the ingress pipeline, the second factor is the strict prioritization of user traffic over IDLE packets of the same port, also introduced in Section 3.1. The second factor, in particular, provides an upper bound on the impact of the IDLE packets as long as the user traffic respects the minimum average frame size requirements of the switch specification (see Appendix D for a formal analysis).

To truly stress these mechanisms, we evaluate an extreme scenario in which multiple hosts send minimum-size (64 B) packets toward a single egress port and OrbWeaver's filtering mechanism is disabled. This situation is not possible in OrbWeaver, but is helpful in demonstrating the efficacy of egress prioritization for protecting user traffic. The results verify the analysis above, even for high user-traffic utilization. For comparison, we also show the impact of an IDLE stream operating at the order-of-magnitude-higher maximum rate of Figure 3 but still set to low-priority. Again, across all experiments, throughput was unaffected.

3.2.4 Does Injection Affect Power Usage?

Finally, we investigate the impact of weaving on the power consumption of today's switches. A natural concern is that the continuous stream of packets will increase consumption; however, we find the actual impact is minimal as the underlying Ethernet MAC already continuously sends IDLE symbols.

To evaluate this, we used a P3 Kill-A-Watt Electricity Usage Monitor (Model P4400) to measure the total power draw of a Wedge100BF-32X programmable switch. The monitor sits between the switch's power plug and its power outlet and can measure wattage to within 0.2–2.0%. To emulate the switch's deployment into a network of programmable switches, we connect every port on the switch to a second switch that logically functions as 32 neighboring switches. We test three distinct configurations:

- *Baseline:* All ports on the switch are connected at 100 Gbps; however, the switch is otherwise inactive, i.e., there is no incoming traffic nor any IDLE packets.
- *Only OrbWeaver:* Same as above, but with OrbWeaver's IDLE stream generation enabled on all switches. The switch is, thus, both sending and receiving packets at T.
- *Maximum utilization:* The worst case scenario, where the switch is both sending and receiving user packets at the maximum rate and generating IDLE packets (that are eventually dropped in the ingress pipeline).

Figure 7 shows the power draw of each configuration over a 1 min period. OrbWeaver's transmission of packets at rate T increases the average power draw of the switch by $<2\%$.

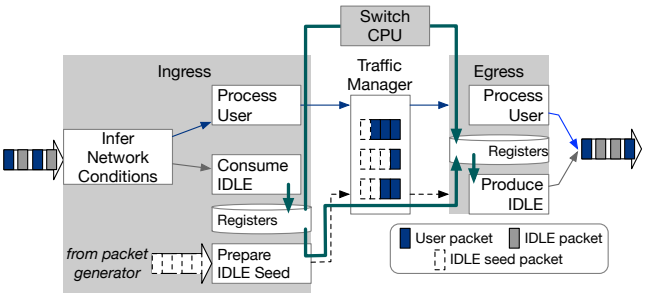


Figure 8: Structure of a P4 program that processes a weaved stream. The ingress pipeline extracts information from the weaved stream, then processes user and IDLE packets separately. The egress pipeline processes user packets and transforms seed packets into IDLE packets. Pipelines can communicate using registers that are synchronized with either seed packets or the switch CPU, as shown by the thick lines.

4 Use Cases

Figure 8 outlines the general structure of a P4 program that uses OrbWeaver. Whereas a standard P4 program processes a stream of user packets, an OrbWeaver P4 program processes a weaved stream of user and IDLE packets. OrbWeaver programs can append/read information from the payloads of the IDLE packets (which appear on the wire as a special EtherType) or infer statistics from the timing of the weaved stream. In either case, the content of IDLE packets can be manipulated just like any other packet (metadata like the drop decision, priority, or egress port should not be changed).

In typical usage, the receiving switch will process, record, and drop incoming IDLE packets before the end of the ingress

pipeline. In most cases, the IDLE packets bypass the normal pipeline logic and, thus, will not affect user byte/drop/error counters. Separately, they use either (a) an agent on the switch CPU [47] or (b) a locally generated IDLE seed packet to transfer data from the ingress to the egress pipeline before sending to the downstream switch. Together, they facilitate multi-hop communication over IDLE packets.

In this section, we detail three example use cases of OrbWeaver (see Appendix A for others). For each example, we consider a recently proposed network system, and we explore how well OrbWeaver can approximate it without introducing any additional impact on user traffic. We note that, in some cases, this restriction can result in suboptimal designs (i.e., imposing on user traffic may result in better overall performance, even if it incurs overhead). Rather, we ask: how far can operators go before needing to ever consider the choice between network throughput and features?

4.1 Use Case #1: Fast Failure Detection

Failures of network components are common in large networks where the number of devices involved ensures a constant flow of incidents. Reasons for the failures include overheating components, power instability, bit flips in the signal, loose transceivers, bent fibers, or any number of other causes [15, 44, 51, 52]. In the end, however, the symptom of many of these failures is the same: lost packets in the network.

Thus, as the first steps toward mitigation, quickly detecting and quantifying packet loss is critical to maintaining high availability and stringent SLOs, particularly as networks improve in both bandwidth and reaction time such that control-plane processing is no longer the sole bottleneck [11, 26, 30–32, 47]. Unfortunately, as mentioned in Section 2, common detection approaches—periodic keepalives or pings—force network architects to sacrifice detection latency to constrain overheads. Moreso as pings are traditionally prioritized over user packets to minimize false positives.

Even recent systems like NetSeer [50] that track user-packet loss inband (without injecting additional packets) suffer from this tradeoff. For example, NetSeer's choice to not inject additional packets means that the network is necessarily slow to detect a black hole (differentiating from a lack of demand requires CPU coordination to compare the flow counters of adjacent switches). Likewise, their choice to tag every packet with a sequence number incurs a bandwidth overhead of 0.3%∼6.3% in return for higher detection granularity (unless there are previously unused bits in the header and we cannot change the data plane to remove them).

4.1.1 An OrbWeaver Redesign

Taking NetSeer as a base, we can replace its inter-switch communication with an OrbWeaver-influenced design to eliminate bandwidth overheads and significantly improve detection time. We refer readers to the original paper [50] for full details of the existing system but summarize the relevant components as follows. NetSeer records the 5-tuple of each packet in the egress pipeline using per-port ring buffers and tags it with a 4-byte sequence number. The downstream switch stores the last observed sequence number. Upon detecting a gap (e.g., packet 14 after packet 12), it sends 3 duplicate and high-priority drop notifications to the upstream switch for each missing sequence number. If the upstream switch receives at least one such notification, it will use the records in the ring buffer to generate a flow event for the missing packet, which will be compressed/summarized for the management plane.

In NetSeer-OW, switches maintain per-port hash tables that, like NetSeer, record the 5-tuples and packet counts of passing flows (using the 5-tuple hash as the index). The caches are maintained in the egress pipeline of each upstream switch as well as the ingress pipeline of each downstream switch. As channels are FIFO and the tables use the same size and deterministic hash function, their content should always be identical. The only exceptions occur after a packet loss, at which point either a counter or a 5-tuple will differ.

In this re-design, user packets are *not* tagged with any additional data nor does it require triple-notifications. Instead, the upstream switch will opportunistically embed in IDLE packets psuedo-randomly selected cache records[2]. If the downstream switch finds that a record differs from its local copy, it will generate an event for the contained 5-tuple. It will also generate an event if packets stop arriving, which is detected with locally generated IDLE seed packets that scan per-port weaved-stream counters. After NetSeer-OW compresses/filters these events, the control plane sends the results over a low-priority TCP connection to the central controller.

Note that, in addition to exploiting the IDLE stream to carry flow information, (R1)'s guarantee of packet arrival rates enables provably optimal detection speed of link failures. In principle, OrbWeaver can trigger an alert if the `ingress_mac_tstamp` of any two consecutive packets is $\leq \tau$. While that level of granularity may be too aggressive for many networks, we note that recent proposals for data plane rerouting have made detection speed a bottleneck [11, 26, 30, 32], particularly if a goal is zero-loss failure recovery. In the end, the point is that OrbWeaver can provide arbitrarily precise failure detection/statistics for current and future networks.

Dealing with a lack of sending opportunities. While extended periods of maximum utilization are rare in most networks [9, 38, 49], NetSeer-OW can still provide useful properties during these extreme conditions. For example, for failure detection, a downstream switch in a fully utilized network can immediately detect a packet drop by examining the gaps between adjacent packets (a drop occurred when the gap $> \tau$).

Flow attribution is slightly more challenging, with the chief concern that the switch evicts the flow before including it in an IDLE packet. We can quantify the probability of this hap-

[2] To improve the update rate, we can pack up to three 5-tuple-counter records (IPv4 and counters of 3 B) in each packet. To handle register access limitations, we can pack the records or split the table across multiple arrays.

	NetSeer		NetSeer-OW	
Data structure size (per-port)	**256**	**64**	**512**	**128**
SRAM (KB)	384	192	896	320
Number of sALU/register arrays	6	6	7	7

Table 1: Data plane resource usage for typical NetSeer and NetSeer-OW configurations on a 64×100 Gbps switch.

pening using the formalization in Appendix E. For reference, using the assumptions of Appendix E, average utilization of [9, 38], and flow cache performance of [39], ISP routers with 128 cache entries per port would have a $P(\text{notified}) \approx \frac{0.72}{0.72+0.28*0.45/3} = 94.4\%$. A data center switch with 128 cache entries would have $P(\text{notified}) \approx \frac{0.75}{0.75+0.25*0.16/3} = 98.2\%$, or with 512 entries $P(\text{notified}) \approx \frac{0.75}{0.75+0.25*0.05/3} = 99.4\%$.

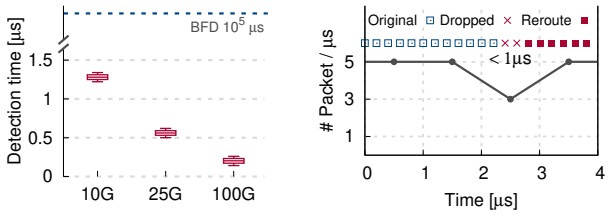
Benefits. Compared to the original NetSeer design, the primary benefit of the OrbWeaver augmentation is to completely eliminate *all* sources of bandwidth overhead—in essence, we can apply NetSeer for 'free.' In particular, it eliminates the overhead of sequence number tagging (0.3%∼6.3%) of capacity; the replicated, high-priority failure notifications (up to 100% of reverse link capacity); and the impact on user traffic of the event reports. Beyond overhead, it also improves the speed to detect inter-switch failures, particularly during periods of low utilization.

Table 1 shows the data plane memory consumption of both systems. Additional memory increases $P(\text{notified})$, however the relationship is different for each system. As a concrete data point, consider the coverage goal highlighted in the original NetSeer evaluation [50]—to correlate 90% of packet loss events with flows. For a 64×100 Gbps switch and a similar estimation strategy as above, NetSeer-OW meets this goal with 320 KB of SRAM (128 cache slots per port) in both ISP and data center workloads. On the other hand, assuming the network's minimum packet size is 64 B, NetSeer requires approximately 384 KB of SRAM to meet the 90% coverage objective because it must allocate enough ring buffer slots per port (256) to ensure that sequence numbers are not overwritten before switches have a chance to correlate their results.

4.1.2 Evaluation

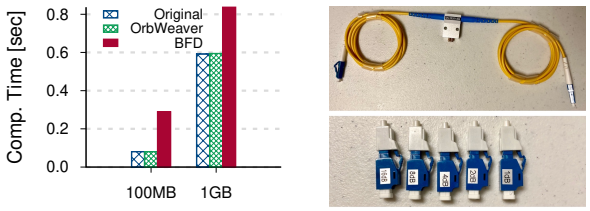
Detecting failures more quickly. To quantify how quickly NetSeer-OW can detect a failure, we deployed NetSeer-OW to a hardware testbed and randomly disconnected a link between the two switches A and B 100 times to emulate 100 fail-stop link failure events. To test the limits of our approach, we configured the probes to mark a τ-timeout failure as soon as even a single packet loss is detected.

Figure 9a shows the detection time of trials for 10, 25, and 100 Gbps links. NetSeer-OW achieved 100% precision and recall. It also consistently detected the failure within 10s of nanoseconds of the optimal time. In contrast, typical configurations for protocols like Bidirectional Forwarding



(a) Link fail-stop detection (b) Link fail-stop recovery

Figure 9: (a) the min, Q_1 (p25), median, Q_3 (p75), and max of OrbWeaver's time to detection across 100 failure events. (b) OrbWeaver's time to recovery (<1 µs) from a bidirectional failure of a 25 Gbps link. A total of two packets are lost.



(a) Completion time during failures (b) Optical attenuators

Figure 10: (a) shows the transfer completion time comparison for original, NetSeer-OW, and BFD (100 ms) in a simple leaf spine topology. With NetSeer-OW's fast detection and data plane reroutes, the impact is minimal.

Detection (BFD) are closer to 10s or 100s of milliseconds; even recent data plane detection systems [20, 30] are several orders of magnitude slower than NetSeer-OW can achieve.

Figure 9b shows the resulting seamless recovery when NetSeer-OW is combined with a simple data plane rerouting mechanism. In the experiment, we induce a bidirectional failure in one link between A and B, and we configure B to failover to a backup path as soon as it detects an error. On top of this setup, we send a steady stream of packets on the target link at a relatively high rate of 5M packets per second. A total of two packets were lost—likely in-flight.

End-to-end impact. To evaluate the end-to-end impact, we emulate a leaf-spine topology with 2 leaf switches L1, L2 and 2 spine switches S1, S2. All switches run OrbWeaver with pre-computed data plane backup paths. Between L1 and L2, we insert a variable fiber optic in-line attenuator capable of 0∼60 dB attenuation. On hosts connected to the leaf switches, we run TCP transfers of varying sizes using `iperf`, during which we increase attenuation from zero until failure and examine the impact over the transfers experiencing the events. As Figure 10a shows, with OrbWeaver, the impact of failure is negligible with respect to completion time. In contrast, with BFD, failures cause the 100MB transfers to take over 4× longer and the 1GB transfers to take over 30% longer.

4.2 Use Case #2: Time Synchronization

Time synchronization is another common task in modern networks. Like failure detection, time synchronization requires

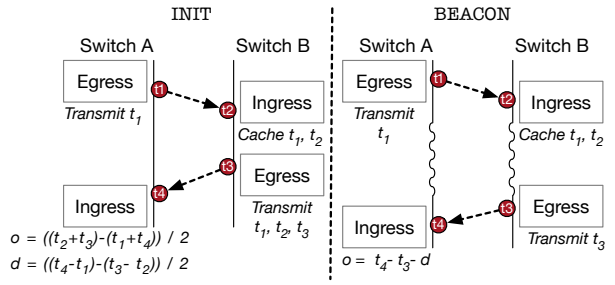


Figure 11: Time sync in DPTP-OW, using IDLE packets. When the difference between t_2 and t_3 is small, A treats the message as part of an `INIT` phase and calculates o, the clock offset, and d, the one way delay. When it is high, the `BEACON` phase uses the most recent d to track clock frequency drift.

coordination between adjacent switches, and many other applications rely on its accuracy [13, 36, 45, 46].

Unfortunately, the most common methods for synchronizing time between adjacent machines involve the computation of One-Way Delay (OWD) using periodic, high-priority echo requests/replies [1, 14, 30]. Here too, architects are presented with a tradeoff: clock frequency drifts imply that the faster we send echoes, the more closely we can bound the clock offset and the more accurate the synchronization. Protocols like DTP [29] that integrate the protocol into the physical layer can circumvent this overhead but require hardware changes.

4.2.1 An OrbWeaver Redesign

The state-of-the-art in time synchronization for programmable switches is DPTP [25]. In it, two adjacent switches (a client, A, and a server, B) compute the offset of their local clocks by leveraging switches' ability to embed timestamps into each packet during different stages of packet processing. Host and multi-hop synchronization are also possible using multiple strata. The protocol calls for three messages in each round of the protocol: (1) a DPTP request $[A \rightarrow B]$, (2) a DPTP response $[B \rightarrow A]$, and (3) a DPTP follow-up $[B \rightarrow A]$. All three messages are high-priority to eliminate queuing delay.

(1) is timestamped using the Tofino `egress_deparser_tstamp` and `ingress_mac_tstamp` of A (t_1) and B (t_2), respectively. (2) is timestamped using the same counters in B (t_3) and A (t_4), respectively. In a traditional clock synchronization protocol, the offset would be computed as $\frac{(t_2+t_3)-(t_1+t_4)}{2}$. Unfortunately, we note a fundamental limitation of today's programmable switches—that the `egress_deparser_tstamp` does not capture the actual point of packet serialization. Thus, the computed offset is subject to variable delays as a result of egress MAC contention. As a result, DPTP introduces the third packet, the follow-up, which embeds a more accurate egress serialization timestamp (obtained out-of-band). Again, we refer interested readers to [25] for full details.

An OrbWeaver-inspired redesign can obviate the need for the third, follow-up message by inferring the egress MAC contention from the weaved stream (and only using results with no contention). This allows us to use the traditional two-way protocol of Figure 11. It can also eliminate the impact of the remaining messages using opportunistic sends.

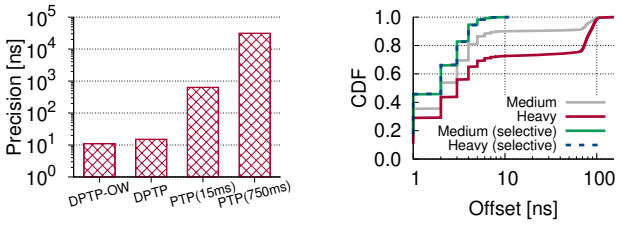
Opportunistic synchronization: Rather than relying on high-priority echoes, a system can rely solely on OrbWeaver's IDLE packets to piggyback timestamps. In particular, whenever A has an opportunity, it sends a request to B on an IDLE packet with a field for t_1. Upon receiving the packet, B maintains a cache for the most recent values of t_1 and t_2. Separately, whenever B has an opportunity, it sends the most recent values of t_1 and t_2 along with the local `egress_deparser_tstamp` in t_3. In an empty network, A can calculate the clock skew as $\frac{(t_2+t_3)-(t_1+t_4)}{2}$ just as DPTP but with much more frequent synchronization (leading to lower *jitter*, i.e., nominal error [33]).

A challenge with the above approach occurs in networks with high utilization. The traditional OWD estimation method used above implicitly assumes that the clock drift is constant for the duration of the protocol round; otherwise, the delays at the time of the request and response may not be comparable due to clock frequency drift. In OrbWeaver, this can happen if there is congestion from B to A; the gap between t_2 and t_3 can be unbounded, leading to inaccurate results.

We address this challenge by borrowing an idea from a different protocol, DTP [29]: the decoupling of synchronization into INIT and BEACON rounds. If the time between t_2 and t_3 is sufficiently small, the round is treated as an INIT round and A computes the offset as above. Otherwise, A treats the message as part of a BEACON round where it takes d, the OWD computed from the last INIT round ($d = \frac{(t_4-t_1)-(t_3-t_2)}{2}$), and it computes a new offset: $o' = t'_4 - t'_3 - d$.

Selective synchronization: Finally, to remove the need for DPTP's third 'follow-up' message, we can exploit the implicit information contained in the woven stream's timing. The underlying intuition is simple: if the gap between an IDLE packet and its preceding packet is less than τ, the IDLE packet may have encountered contention at the egress MAC. In this case, the packet's timestamp may be unreliable. DPTP corrects for this contention with the follow-up message; OrbWeaver simply ignores these protocol rounds. While this filtering effectively requires that usable gaps be $> \tau \sim 2\tau$, it greatly improves the accuracy of the protocol while still permitting frequent re-synchronization in modern networks.

Dealing with a lack of opportunity to send. The above protocol fully synchronizes switches when both links have concurrent IDLE gaps. The protocol also includes support for correcting small drifts when only one direction has a gap (by adjusting to the fastest clock in the network). We note that in a network with multiple paths, we can configure synchronization to propagate among any one of those paths. Thus, if we view the network as a directed graph, the only time a switch may lose synchronization is if sufficient links are maintaining 100% utilization that the links form a cut of the graph. In the end, if operators need assurances, they may need to send



(a) Clock precision (b) Clock jitter

Figure 12: (a) shows the precision for different synchronization protocols and a heavy workload (∼80% CAIDA user traffic). (b) shows the CDF of observed offsets (absolute value) for DPTP-OW upon medium and heavy loads for 10 Gbps link ($\tau = 1200$ ns), w/ or w/o selective sync. OrbWeaver achieves a precision of 11 ns even under heavy user traffic.

higher-priority messages if too much time elapses; however, we can extend our techniques so that the messages only need to be prioritized above the lowest-priority user traffic—high-priority, interactive applications would be unaffected.

Benefits. As long as there is occasional usable bandwidth in the network, OrbWeaver again eliminates all bandwidth overheads without sacrificing accuracy or nominal error. When the network is underutilized, it actually provides similar re-sync intervals as DTP but using commodity PISA switches.

4.2.2 Evaluation

Following prior work, we evaluate DPTP-OW's precision [29, 43] (defined as the maximum clock skew in the network), as well as its jitter [33] (defined as the distribution of measured offsets or nominal error). Again to match prior work, we evaluate these in a two-switch testbed during a 20 min collection for 10 Gbps link with a medium workload (a CAIDA trace with 25% average utilization) and a heavy workload (the same trace sped up to ∼80% average utilization). We compare to both DPTP (with 2000 requests/sec) and PTP. For PTP, prior work has suggested message frequencies ranging from 15 ms to 2 s [1, 2, 29, 30]; we pick two points in this range: 15 ms as a lower bound and 750 ms per the evaluation baseline [29].

We observe that, even at high loads, DPTP-OW can achieve 10 ns bounds in both precision (Figure 12a) and jitter (Figure 12b) without imposing on user traffic. These bounds are similar to or better than DPTP, which incurs high-priority bandwidth overhead. Preliminary tests on higher-link speeds indicate that precision will only improve as τ decreases. In Figure 12b, we further observe that selective synchronization is an effective technique to reduce the message complexity of the protocol while maintaining low jitter and good precision.

4.3 Use Case #3: Congestion Feedback

Finally, many modern networks rely on robust load balancing algorithms to efficiently utilize their multiple paths. There are numerous approaches to load balancing, but among them, adaptive approaches [3, 26] are attractive as they can react to current network conditions when making balancing decisions.

A state-of-the-art approach is taken by HULA [26], which proposes a protocol for adaptive data center load balancing using programmable switches. In HULA, every switch maintains two tables: a `bestHop` table that stores the best next-hop to each destination ToR, and a `pathUtil` table that stores the utilizations of those next-hops. Destination ToRs periodically flood the network with high-priority probes that traverse all paths (in the reverse direction, dst-to-src) and track the bottleneck link utilization of the best such path—intermediate switches update their `bestHop`/`pathUtil` tables accordingly.

As in the previous use cases, congestion feedback mechanisms like the one in HULA force a tradeoff between overhead and the availability/freshness of congestion data. HULA eventually sets the probing interval to 1-RTT and makes a case for why that is a good tradeoff, but OrbWeaver can potentially provide similar performance using only opportunistic sends.

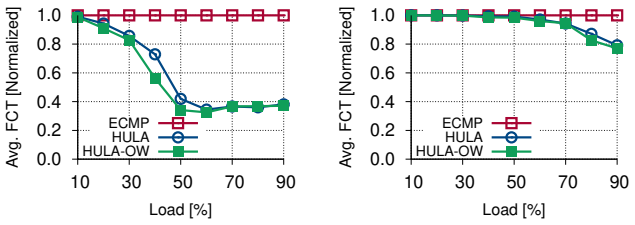
4.3.1 An OrbWeaver Redesign

An OrbWeaver-inspired redesign replaces the high-priority HULA probes with OrbWeaver's opportunistic IDLE packets. There are two new challenges. The first is building a 'flood' communication model on top of OrbWeaver's opportunistic sends. The second is dealing with congestion on the reverse path and the resulting lack of new information.

Per-path propagation: For any path through the network, there are two types of hops: ingress-to-egress hops (that bridge the pipelines of a local switch) and egress-to-ingress hops (that bridge adjacent switches).

For the former, HULA-OW leverages the switching ASIC's PCIe interface to asynchronously mirror the `pathUtil` table between the ingress and egress pipelines of a single switch. We use Mantis [47] to mirror the registers, which completes a mirror operation every ∼20 µs without impacting data plane throughput. For the latter type of hop, the system simply sends the contents of `pathUtil` using IDLE packets. To make this process more efficient, we can stripe the `pathUtil` table across m registers and pack m `(dstToR, pathUtil)` records into each IDLE packet round-robin style. In an unloaded network, the full table is transmitted in $\frac{R\tau}{m}$ time, where R is the number of ToRs in the data center. We note that even for $R = 1000$ and $m = 1$ (i.e., an unoptimized update rate), this is still more frequent than HULA.

Stale information: If there is persistent congestion on the reverse path, utilization information may not be able to propagate across the network; the switch adjacent to the congestion will know the utilization of the adjacent link, but not downstream links. To handle this case, HULA-OW uses a simple aging mechanism. Specifically, it will track the EWMA of all observed `pathUtil` values for every destination ToR (in addition to the minimum). After each RTT with no information from the best path, it will shift the best path's `pathUtil` value toward the average (with a lower bound of the adjacent link's



(a) Web Search (b) Data Mining

Figure 13: Avg. FCT (normalized to ECMP) for HULA and HULA-OW upon different loads of DCTCP and VL2 traces.

utilization). If no information comes from *any* neighbor for several RTT and the adjacent links are all equal, the switch will fall back to random flowlet placement.

Dealing with a lack of opportunity to send. We note that the effect of the above metric-aging strategy is that `bestHop` will be quickly overwritten by the 'next-best hop' whose reverse path has opportunities to send. Assuming that at least some congestion information gets through, HULA-OW will still provide substantial benefits due to properties like the power of two choices [34]. In the worst case, it achieves equivalent performance to flowlet ECMP.

Benefits. Across all regimes, HULA-OW eliminates the probe overhead on network bandwidth. In networks with low utilization or high burstiness, it provides more frequent utilization updates than HULA in addition to increasing the peak usable bandwidth (see below).

4.3.2 Evaluation

Performance. We evaluate HULA-OW in NS-2 using the same FatTree topology as the original paper (Figure 4 of [26]). Also like HULA, we leverage synthetic workloads based on web-search [4] and data-mining [16]) and configure HULA to probe at a 200 µs interval. Figure 13 shows the avg. FCT (normalized to ECMP) for HULA and HULA-OW.

Despite the frequent periods of full utilization in these workloads (especially at high average load), we observe that HULA-OW is able to find sufficient gaps between packets to efficiently transfer utilization information. Overall, HULA-OW is able to provide comparable or better performance than HULA in all of the tested cases, even in the presence of very high average utilization. The performance is also always either equivalent or better than the ECMP baseline.

Overhead reduction. The bandwidth overhead of HULA probes is given by $\frac{probeSize \times numToRs \times 100}{probeFreq \times linkBandwidth}$ [26]. With 500 ToRs, probeFreq=200 µs, probeSize=64 B, and 100 Gbps links, HULA occupies 1.6% of the network's bandwidth. In contrast, HULA-OW occupies close to *zero* of the network's usable bandwidth and only 1.5% of the packet-level capacity of the ingress pipeline (which HULA's probes also consume).

5 Related Work

Leveraging unused resources. OrbWeaver is not the first system to propose the opportunistic use of leftover resources. Indeed, many applications of priorities are in a similar spirit. Even in contexts outside of computer networking, others have used low-priority background tasks and spot VMs to harvest unused CPU cycles and memory [5].

In networking, close related work includes software WANs like SWAN [21] and B4 [24], which divide traffic into classes that range from interactive to background—interactive traffic is given priority while background traffic soaks up any remaining bandwidth. These systems successfully provide opportunistic bandwidth utilization but focus on end-host data. As explained in Section 2, these approaches can leave parts of the network unutilized due to both application traffic patterns and structural bottlenecks. OrbWeaver is, thus, complementary to these approaches and can be used to reclaim the remaining bandwidth for intra-network coordination.

Prior work has also applied similar techniques to lower layers, for instance, in the case of Ethernet's IDLE symbols or F10's rapid heartbeats [32]. F10, in particular, proposed a failure detection mechanism that is close to OrbWeaver's in which devices continue to send traffic even when idle. In comparison, OrbWeaver's contribution is make the idea practical on high-speed programmable switches, to closely examine the resulting impacts on switch configurations and user traffic, and to show how to seamlessly integrate the weaved stream into a spectrum of applications beyond the use case of F10.

Applications of OrbWeaver. OrbWeaver also builds explicitly on prior work that improves networks with coordination, signaling, and probes. We refer readers to the relevant parts of Section 4 for a discussion of the systems on which OrbWeaver builds, and to the original papers for a more complete examination of related work for our applications.

In general, however, OrbWeaver improves on much of the prior work by providing comparable or better performance with near-zero overhead. Exceptions include systems like F10 [32] and DTP [29], which use hardware support to eliminate protocol overheads. As mentioned above, OrbWeaver's contribution is to generalize the concept and demonstrate a practical framework for it on commodity network devices.

6 Conclusion

Must data plane applications always choose between coordination fidelity and bandwidth overhead? This paper demonstrates that, somewhat surprisingly, they do not. To that end, we introduce OrbWeaver, a framework for opportunistic coordination in a manner that does not affect user traffic or switch power consumption. Using three recently proposed systems, we show how to leverage OrbWeaver to eliminate their bandwidth overheads while maintaining their efficacy.

Acknowledgments

We gratefully acknowledge Vladimir Gurevich for his assistance in understanding the Tofino switch architecture. We also thank Vladimir, Gianni Antichi, our shepherd Aurojit Panda, and the anonymous NSDI reviewers for all of their thoughtful comments. This work was funded in part by Google, Facebook, VMWare, and NSF grant CNS-1845749.

References

[1] Ieee standard 1588-2008. https://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4579757, 2008.

[2] Juniper precision time protocol overview. https://www.juniper.net/documentation/us/en/software/junos/time-mgmt/topics/concept/ptp-overview.html, 2020.

[3] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 503–514, 2014.

[4] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[5] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. Providing slos for resource-harvesting vms in cloud platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751. USENIX Association, November 2020.

[6] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang (Harry) Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 419–435, Renton, WA, April 2018. USENIX Association.

[7] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. SIGCOMM '20, page 662–680, New York, NY, USA, 2020. Association for Computing Machinery.

[8] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.

[9] Caida. The caida ucsd statistical information for the caida anonymized internet traces. https://www.caida.org/data/passive/passive_trace_statistics.xml, 2019.

[10] M. Chiesa, R. Sedar, G. Antichi, M. Borokhovich, A. Kamisiński, G. Nikolaidis, and S. Schmid. Fast reroute on programmable switches. *IEEE/ACM Transactions on Networking*, pages 1–14, 2021.

[11] Marco Chiesa, Roshan Sedar, Gianni Antichi, Michael Borokhovich, Andrzej Kamisiński, Georgios Nikolaidis, and Stefan Schmid. Purr: A primitive for reconfigurable fast reroute: Hope for the best and program for the worst. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 1–14, 2019.

[12] Intel Corporation. P4-16 intel tofino native architecture – public version. Application Note 631348-0001, Intel Corporation, March 2021.

[13] Thomas G. Edwards and Warren Belkin. Using sdn to facilitate precisely timed actions on real-time data streams. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, page 55–60, New York, NY, USA, 2014. Association for Computing Machinery.

[14] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 81–94, 2018.

[15] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 conference*, pages 350–361, 2011.

[16] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 51–62, 2009.

[17] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 139–152, 2015.

[18] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 357–371, New York, NY, USA, 2018. Association for Computing Machinery.

[19] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 71–85, 2014.

[20] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast connectivity recovery entirely in the data plane. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 161–176, 2019.

[21] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 15–26, New York, NY, USA, 2013. Association for Computing Machinery.

[22] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. Contra: A programmable system for performance-aware routing. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 701–721, 2020.

[23] Van Jacobson. Compressing tcp/ip headers for low-speed serial links. Technical report, RFC 1144, February, 1990.

[24] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 3–14, New York, NY, USA, 2013. Association for Computing Machinery.

[25] Pravein Govindan Kannan, Raj Joshi, and Mun Choon Chan. Precise time-synchronization in the data-plane using programmable switching asics. In *Proceedings of the 2019 ACM Symposium on SDN Research*, pages 8–20, 2019.

[26] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, pages 1–12, 2016.

[27] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *Demo paper at SIGCOMM '15*, 2015.

[28] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. Redplane: Enabling fault-tolerant stateful in-switch applications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 223–244, New York, NY, USA, 2021. Association for Computing Machinery.

[29] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 454–467, 2016.

[30] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter H Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, et al. Sundial: Fault-tolerant clock synchronization for datacenters. 2020.

[31] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. Ensuring connectivity via data plane mechanisms. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 113–126, 2013.

[32] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 399–412, 2013.

[33] Jim Martin, Jack Burbank, William Kasch, and Professor David L. Mills. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905, June 2010.

[34] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[35] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 85–98, 2017.

[36] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized "zero-queue" datacenter network. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 307–318, New York, NY, USA, 2014. Association for Computing Machinery.

[37] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 123–137, 2015.

[38] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *SIGCOMM Comput. Commun. Rev.*, 45(4):183–197, August 2015.

[39] John Sonchack. *Balancing Performance and Flexibility in Hybrid Network Telemetry Systems*. PhD thesis, University of Pennsylvania, 2020.

[40] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. Lucid: A language for control in the data plane. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 731–747, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Charles E. Spurgeon. *Ethernet: The Definitive Guide*. O'Reilly & Associates, Inc., USA, 2000.

[42] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. Flightplan: Dataplane disaggregation and placement for p4 programs. In *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21)*, pages 571–592, 2021.

[43] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017.

[44] Xin Wu, Daniel Turner, Chao-Chih Chen, David A Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: automating datacenter network failure mitigation. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 419–430, 2012.

[45] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 402–416, 2018.

[46] Nofel Yaseen, John Sonchack, and Vincent Liu. tpprof: A network traffic pattern profiler. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 1015–1030, Santa Clara, CA, February 2020. USENIX Association.

[47] Liangcheng Yu, John Sonchack, and Vincent Liu. Mantis: Reactive programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 296–309, 2020.

[48] Lior Zeno, Dan RK Ports, Jacob Nelson, and Mark Silberstein. Swishmem: Distributed shared state abstractions for programmable switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 160–167, 2020.

[49] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, pages 78–85, 2017.

[50] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. Flow event telemetry on programmable data plane. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 76–89, 2020.

[51] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 479–491, 2015.

[52] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 362–375, 2017.

Application Class	System	Weaved Inference?	IDLE Messaging?	Description
Traffic Engineering	Flowlet load balancing [3, 26]	✓	✓	Section 4.3.
	Performance-aware routing [22]		✓	Propagate route updates in customizable distance-vector routing algorithms using IDLE packets.
	Micro-burst detection [49]	✓	✓	Detect micro-bursts from weaved stream, provide feedback to upstream switches with IDLE packets.
Fault Tolerance	Fast failure recovery [50]	✓	✓	Detect failures (Section 4.1), alert upstream switches with IDLE packets for fast data-plane mitigation [10].
	Consistent replicas [28, 48]		✓	Synchronize eventually-consistent distributed state, e.g., for distributed firewalls, with IDLE packets.
Monitoring	Packet forensics [19]		✓	Transfer packet postcards in IDLE packets to reduce overhead of packet history tracking.
	Network queries [18, 35]	✓	✓	Support queries over both flow and weaved stream statistics, export query results in IDLE packets.
	Latency localization [17]	✓	✓	Measure latency in network core using weaved stream, disseminate measurements with IDLE packets.
Network Services	Clock synchronization [25]	✓	✓	Section 4.2.
	Header compression [23, 42]		✓	Synchronize state of point-to-point packet header compressors with IDLE packets.
	Event-based network control [40]		✓	Carry network control events in IDLE packets.

Table 2: OrbWeaver use cases. A diverse range of data-plane applications can use OrbWeaver's weaved stream to learn about conditions in the network and/or communicate via IDLE packets that consume no data-packet bandwidth.

A Applications of OrbWeaver

Table 2 surveys 11 applications that can benefit from an OrbWeaver implementation, belonging to four distinct classes. We describe several implementations in Section 4. All applications can be expressed as OrbWeaver P4 programs with the basic architecture shown in Figure 8.

Across all applications, we find that there are two overarching benefits to an OrbWeaver implementation:

1. OrbWeaver's weaved stream allows data plane applications to infer information about network conditions, such as the presence of congestion or failures in an upstream path.
2. OrbWeaver's IDLE packet abstraction lets data plane applications disseminate information without consuming user bandwidth. IDLE packets are useful for data transfer between directly connected switches (e.g., to synchronize the context tables of a switch-to-switch packet-header compressor [42]) or across the wider network (e.g., to disseminate information about network faults [32], congestion [49], or even user query metrics [35]).

We note that our focus of these applications and this paper is in-network communication. However, end hosts may also be able to benefit from OrbWeaver, e.g., by examining the output of the weaved stream coming from host-facing ports of ToR switches. Efficient end-host generation of a weaved stream may also be possible, but we leave a full exploration to future work.

A.1 Balancing Multiple Applications

IDLE packets are generated and weaved entirely by the OrbWeaver framework. Applications only embed information and extract it in the receiver. IDLE packets can carry the information of multiple applications. For example, a time synchronization application that needs 12B to carry 4 timestamps can co-exist with a failure detection protocol that needs 48B. In this paper, we assume minimum-sized packets but, in principle, IDLE packets can be MTU-sized with the only effect being a proportionally increased worst-case packet delay. Of course, there are fundamentally a limited number of bytes in each IDLE packet; OrbWeaver leaves the decision on how to allocate these bytes to network architects and operators.

A.2 Preventing Starvation

The primary goal of the paper is to explore the opportunistic use of IDLE cycles for in-network coordination. Because of our opportunistic approach, there may be cases where IDLE packets get starved by user packets; however, as previously noted, two factors mitigate the issue:

- The lack of IDLE packets itself reveals concrete information of the network condition (per R1 guarantee of the weaved stream predictability).
- Prior works observed that persistent user traffic is rare, instead, IDLE cycles (every 10s or 100s of µs) are ubiquitous.

A wide range of applications can be implemented with only opportunistic communication. Of course, some applications may need additional guarantees, e.g., applications requiring a strict, real-time guarantee w.r.t. minimum rate (i.e., maximum inter-IDLE-packet gap); or applications that need more aggregate bandwidth than the weaved stream can guarantee in a timely fashion.

In these cases, networks can apply a priority escalation mechanism by adding a single register of N (number of ports) slots and check the elapsed time since last seen IDLE packet. Applications can seamlessly escalate the priority of IDLE packets when too much time passes (per the applications' guaranteed rate SLO). In these situations, OrbWeaver still eliminates nearly all overhead in the presence of (micro)bursts, but may impose a fixed overhead during extended periods of congestion.

B Generalization to Other Platforms

Our focus in this paper was on the Tofino family of programmable switches. While a detailed taxonomy and analysis of every programmable platform is out of the scope of this paper, there is reason to believe that other programmable platforms have similar features or can emulate the features needed to implement OrbWeaver.

In particular, OrbWeaver leverages three hardware features of Tofino switches: (1) packet generation, (2) multicast, and (3) packet prioritization. Among these, support for the latter two can be found in almost every modern forwarding device that is designed to handle the Ethernet protocol. Support for onboard packet generation is not as universal; however, one potential solution is to connect a port on each switch to a simple device/CPU responsible for generating regular, periodic packets. Of course, a CPU, even with real-time scheduling optimizations, may not be as dependable as the Tofino packet generator. This may necessitate additional tolerances.

Finally, our conversations with switch vendors indicate that OrbWeaver's mechanisms will scale to future switches with both increased bandwidth and port counts. Part of this is due to the fact that most of OrbWeaver's components scale with the clock rate of the switch and/or are independent to each pipeline. The notable exception is packet generation; however, we note that OrbWeaver currently has more than an order of magnitude of headroom (Section 3.2.1). If MTU transmission time does eventually outpace packet generation latency, OrbWeaver's properties will degrade gracefully.

C Energy-Efficient Ethernet (EEE)

The Ethernet standard contains an optional EEE mechanism [41], which allows switches to transition links into a Low-Power Idle (LPI) mode when there is no data to send.

OrbWeaver may be able provide compatibility by turning off the IDLE stream on a per-port, per-direction basis if there is no user traffic during the past S seconds. Each packet flowing between two OrbWeaver switches would then need a single bit reserved as an 'LPI' indicator. Upon receiving an IDLE packet with the 'LPI' indicator set, a receiver will change its expectation from requiring a packet every τ_i seconds to requiring one every τ'_i seconds ($\tau'_i \gg \tau_i$). The very first user packet after the low-power idle mode will be sent with the 'LPI' indicator unset. Loss can be addressed by again emulating EEE and sending several indicator packets in a row.

Enabling this feature may impact the responsiveness of OrbWeaver applications, but we note that all of the use cases studied can make do with less frequent but still regular coordination. OrbWeaver may be able to synchronize these low-power updates with existing synchronization-maintenance events in the PHY.

D Proof of Priority-effect on User Traffic

Theorem. For an arbitrary user packet size distribution and arrival process, with strict priority scheduling and a measurement time window $T \gg \Delta t$ (Δt denotes transmission time of a single IDLE packet), the throughput of the user traffic is unaffected by the IDLE stream.

Proof. Consider a packet sequence $p_1, \ldots, p_n$ with size $\Delta t_1, \ldots, \Delta t_n$ and original schedule $t_1, \ldots, t_n$, denote the new schedule upon the coexistence of IDLE stream as $t'_1, \ldots, t'_n$.

We first prove $\forall i \in [1, n-1], t'_i \leq (t_i + \Delta t) \rightarrow t'_{i+1} \leq (t_{i+1} + \Delta t)$. The case for preemptive scheduling is trivially true. We focus on the case of non-preemptive scheduling.

Base case with p_1: the worst case delay of the transmission is when right at t_1, an IDLE packet is scheduled to transmit and with strict priority p_1 is scheduled right next to it. Hence $t'_1 \leq (t_1 + \Delta t)$.

For the inductive step, given the new schedule of p_i satisfying $t'_i \leq (t_i + \Delta t)$, we need to show that $t'_{i+1} \leq (t_{i+1} + \Delta t)$. There are three cases for the next packet p_{i+1}:

- $t_{i+1} > (t_i + \Delta t_i + \Delta t)$: at t_{i+1}, the previous packet has finished transmission in the new schedule since $t_{i+1} > (t_i + \Delta t_i + \Delta t) \geq t'_i + \Delta t_i$. The worst case delay is when IDLE packet is scheduled right at t_{i+1} and the transmission is delayed by Δt, i.e., $t'_{i+1} \leq (t_{i+1} + \Delta t)$ holds.
- $t'_i + \Delta t_i \leq t_{i+1} \leq (t_i + \Delta t_i + \Delta t)$: at t_{i+1}, p_i finishes transmitting in the new schedule, similar to the previous case, the worst case is Δt when right at t_{i+1}, IDLE packet gets scheduled, hence $t'_{i+1} \leq (t_{i+1} + \Delta t)$ holds.
- $t_i + \Delta t_i \leq t_{i+1} < t'_i + \Delta t_i$: p_{i+1} has been queued since p_i is still transmitting until $t'_i + \Delta t_i$ in the new schedule. With strict priority, p_{i+1} will start transmission right at $t'_i + \Delta t_i$ ignoring the IDLE packet. Hence, $t'_{i+1} = t'_i + \Delta t_i \leq t_i + \Delta t + \Delta t_i \leq (t_{i+1} + \Delta t)$.

Configuration	SRAM	TCAM	Metadata	Tbls	Regs
16×100 Gbps	80 KB	1.28 KB	85 b	3	1
32×25 Gbps	80 KB	1.28 KB	53 b	3	1

Table 3: Additional data plane resources for OrbWeaver's weaved stream generation over an L2 forwarding switch. Ports are binned into groups of 2 and 4, and only 256 multicast groups reserved.

By induction, we have $t'_n \leq (t_n + \Delta t)$, that is, the latency impact is tightly bounded by Δt for an arbitrary user packet and won't accumulate across packets. Given such fixed workload, consider the impact of the IDLE stream over the original transmission time $T = t_n + \Delta t_n - t_1$. For the new transmission time window $[t'_1, t'_n + \Delta t_n]$, the duration $T' = t'_n + \Delta t_n - t'_1 \leq \max(t'_n) + \Delta t_n - \min(t'_1) \leq t_n + \Delta t + \Delta t_n - t_1$. Hence, $T' - T \leq \Delta t$. Since $T \gg \Delta t$, the throughput of the high priority user packet stream is not impacted. □

E Probability of Notification in Use Case #1

We can formally express the probability that a notification is sent before the flow is evicted. Consider the case where there is a drop in flow f and user packets are all MTU-sized, i.e., there is one packet per period, τ. Assume that the flow cache holds N records and 3 can be packed in each IDLE.

$$
\begin{aligned}
P(\text{notified}) &= \frac{P(\text{IDLE contains } f)}{P(\text{IDLE contains } f) + P(\text{new } f' \text{ replaces } f)} \\
&= \frac{\frac{3}{N}P(\text{IDLE})}{\frac{3}{N}P(\text{IDLE}) + \frac{1}{N}(1 - P(\text{IDLE}))P(\text{new flow})} \\
&= \frac{P(\text{IDLE})}{P(\text{IDLE}) + (1 - P(\text{IDLE}))P(\text{new flow})/3}
\end{aligned}
$$

where $P(\text{IDLE})$ is the probability that an IDLE packet was sent during a given period τ, and $P(\text{new flow})$ is the probability that a user packet's flow cannot be found in the cache. Smaller packets multiply the second term in the denominator; a larger N decreases it by improving cache hit rates. The probability that a flow record is evicted *before* it is sent (i.e., that we miss the loss) is 1 less the above value.

F OrbWeaver Data Plane Resource Overhead

Section 3 details the overhead of OrbWeaver's weaved stream generation on user traffic and energy usage. We note that OrbWeaver also uses data plane resources for IDLE seed packet filtering and replication, as shown in Table 3. For each category, OrbWeaver only occupies a small fraction of the total switch resources (for instance $< 1\%$ of both SRAM and TCAM).