BurstRadar: Practical Real-time Microburst Monitoring for Datacenter Networks

Raj Joshi
National University of Singapore
rajjoshi@comp.nus.edu.sg

Ting Qu
National University of Defense Technology
National University of Singapore
quting14@nudt.edu.cn

Mun Choon Chan
National University of Singapore
chanmc@comp.nus.edu.sg

Ben Leong
National University of Singapore
benleong@comp.nus.edu.sg

Boon Thau Loo
University of Pennsylvania
boonloo@seas.upenn.edu

ABSTRACT

Microbursts can degrade application performance in datacenters by causing increased latency, jitter and packet loss. The detection of microbursts and identification of the contributing flows is the first step towards mitigating this problem. Unfortunately, microbursts are unpredictable and typically last for 10's or 100's of μs and the high line rates (> 10 Gbps) in modern datacenter networks further exacerbate the problem. In this paper, we show that modern programmable switching ASICs have made it practical to detect and characterize microbursts at high line rates. Our system, called *BurstRadar*, operates in the dataplane and monitors microbursts by capturing the telemetry information for only the packets involved in microbursts. We have implemented a prototype of BurstRadar on a Barefoot Tofino switch using the P4 programming language. Our evaluation on a multi-gigabit testbed using microburst traffic distributions from Facebook's production network shows that BurstRadar incurs 10 times less data collection and processing overhead than existing solutions. Furthermore, BurstRadar can handle simultaneous microburst traffic on multiple egress ports while consuming very few resources in the switching ASIC.

APSys '18, August 27–28, 2018, Jeju Island, Republic of Korea

ACM ISBN 978-1-4503-6006-7/18/08…$15.00
https://doi.org/10.1145/3265723.3265731

ACM Reference Format:
Raj Joshi, Ting Qu, Mun Choon Chan, Ben Leong, and Boon Thau Loo. 2018. BurstRadar: Practical Real-time Microburst Monitoring for Datacenter Networks. In *9th Asia-Pacific Workshop on Systems (APSys '18), August 27–28, 2018, Jeju Island, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3265723.3265731

1 INTRODUCTION

Over the last decade, the performance of datacenter networks has improved significantly [29]. However, *availability* and service-level guarantees still continue to be challenging problems as datacenter bandwidths increase and applications become more sophisticated [31]. To achieve more 9's in availability and service-level guarantees, we need greater visibility into the network. Modern datacenter networks operate at high-speeds (> 10 Gbps) and have ultra-low end-to-end latency (~10's of μs) [11]. As a result, even small amounts of queuing, called *microbursts*, that occur for short periods of time can have a significant impact on application performance and thereby revenue [1].

Microbursts are events of intermittent congestion that last for 10's or 100's of μs. They increase latency and cause network jitter and packet loss in data center networks [28]. Common causes include TCP Incast scenarios [1], bursty UDP traffic from an offending flow, as well as TCP segment offloading or application-level batching [19]. The performance degradation arising from microbursts is becoming more common today because link speeds are moving beyond 10 Gbps while switch buffers remain shallow. Traditionally, the impact of microbursts has been greatest for high frequency trading (HFT) applications with reported profit differentials of $100 million per year due to latency advantage of just 1 ms [21]. However, today with low end-to-end latency in datacenters and high SLA requirements by applications, the impact of microbursts is no longer limited to such niche applications.

Popular webservices like LinkedIn are reported to have experienced high application latency due to microbursts [18]. To address this problem, we first need to be able to accurately detect the occurrence of these microbursts and identify the contributing flows.

The extremely low timescales make it impossible for traditional sampling-based techniques such as Netflow [8] and sFlow [26] to even detect the occurrence of microbursts. Some existing commercial solutions [7, 17, 24] are able to detect microbursts, but provide no information about the cause. Recent advances in programmable dataplanes [4] and dataplane telemetry have led to proposals for In-Band Telemetry (INT) [13, 20] that embed telemetry information into each packet and enable debugging for several network issues including microbursts. However, since microbursts are unpredictable [32], it is wasteful to use INT to monitor them as it would require the telemetry information for every single packet in the network to be captured and processed, while only a small number of packets contribute to microbursts.

In this paper, we demonstrate that programmable dataplanes can be used to detect microbursts more efficiently by capturing the telemetry information of only the packets involved in microbursts. Our system, called *BurstRadar*, builds on the out-of-band approach for exporting telemetry information [14]. Our key insight is that microbursts are localized to a port's egress queue. This makes all the information required for detecting and characterizing a microburst available *together* on a single switch. Unlike an INT-based approach [13], by detecting a microburst directly on the switch where it happens, we can avoid the computations and delays arising from having to correlate monitoring information from different points in the network. BurstRadar uses a *Snapshot algorithm* (§3.1) to capture information of the packets involved in a microburst. It then use egress packet cloning (§3.2) to generate on-demand *courier* packets for transporting this information together.

While our approach is relatively straightforward given existing programmable dataplane architectures, we made three observations from our design and implementation. First, we need a strategy to temporarily store telemetry information before it can be transferred to the courier packets. Second, there is a sizable delay in the generation of courier packets and it depends on packet size, among other factors. Third, it is possible that if there are multiple simultaneous microbursts on different egress ports, telemetry information for some packets involved might be lost. BurstRadar provisions the temporal storage by implementing a ring buffer using the transactional stateful memory available in the dataplane (§3.3). We then handle the issues of courier packet delays and multiple simultaneous microbursts by sizing the ring buffer appropriately (§3.3).

We have implemented BurstRadar on a Barefoot Tofino [25] switch and evaluated it on a multi-gigabit hardware testbed using utilization burst distributions from Facebook's production network [32]. Our results show that even with microbursts occurring as frequently as every 200 μs, BurstRadar processes 10 times less telemetry information compared to INT [13, 20], while providing all information to fully characterize microbursts and identify contributing flows. BurstRadar captures telemetry information for all packets contributing to microbursts, even with bursts occurring simultaneously on multiple egress ports. Further, it achieves real-time detection of microbursts at multi-gigabit link speeds.

2 RELATED WORK

Commercial solutions such as Cisco's Nexus 5600 and 6000 series switches, as well as Arista's 7150S series switches can detect the occurrence of microbursts but provide no details about the cause [7, 24]. Learning the cause requires traffic mirroring and data correlation across different monitoring data streams [7]. In contrast, BurstRadar provides a full snapshot of telemetry information about the packets involved in a microburst. With this information, we can identify the contributing flow(s) without the significant costs associated with data correlation and traffic mirroring.

In-band Telemetry (INT) [13, 20] is a network debugging tool that is built on programmable dataplanes. INT adds network telemetry information into an additional packet header (INT header) to identify and characterize microbursts. However, since microbursts are unpredictable [32], INT would need to be enabled for all flows in order to detect and characterize microbursts reliably. INT would then need to process telemetry information for every single packet in the network, even though only a small number of these packets contribute to the microbursts. Expensive data correlation would then be required to reconstruct a microburst event. Furthermore, enabling INT on all flows would consume 10% additional bandwidth[1] in the entire network due to the extra INT header. BurstRadar, on the other hand, processes telemetry information only for the packets involved in microbursts, does not require expensive data correlation and is non-intrusive to production traffic since it operates out-of-band. Marple [23] is another network monitoring system that proposes augmenting dataplane programmability with a custom key-value store hardware primitive. It presents a microburst detection case study in which microbursts are assumed to occur at regular intervals. The proposed approach will not work in practice because microbursts do not occur at regular intervals [32]. BurstRadar does not make any such assumption about the arrival pattern of microbursts. It may

[1]For a 5-hop diameter network, INT requires extra 54 bytes per packet [16] which is 10% extra for a median packet size of 500 bytes [2].

be possible to orchestrate BurstRadar's techniques in the Marple framework with some modifications. However, unlike BurstRadar, the hardware primitives required by Marple are not available on today's programmable switching ASICs.

Chen et al. recently proposed Snappy, a technique to estimate the contents of a microburst queue and identify the culprit (heavy) flows in the dataplane [6]. Snappy's detection of culprit flows is however probabilistic in nature and the probability of identifying all the culprit flows (*Recall*) increases with the number of switch pipeline stages used by Snappy. Snappy is expected to require more than 128 stages for achieving a decent Recall in practice[2]. Today's programmable switching ASICs do not currently support such a large number of pipeline stages and we believe that they are unlikely to be available in the near future due to cost concerns. Snappy further requires division and rounding operations which are not currently supported on high-speed programmable switching ASICs. BurstRadar, on the other hand, requires only a modest amount of resources (§4.3) and can thus be implemented on programmable switches available today. The microburst information exported by BurstRadar goes beyond accurate culprit detection and provides a full characterization of microbursts, which is important to network operators for network planning. While Snappy's approach of detecting culprit flows in the dataplane is more suited for automatic microburst mitigation, further work is required to make it practical.

NetSight [14] employs mirroring or packet cloning for exporting telemetry information for *every single* packet traversing a switch. Since packets involved in microbursts form a very small fraction of the overall packets, such an approach is grossly inefficient and infeasible for large datacenter networks. Everflow [33] uses "match and mirror" to selectively trace *specific* packets across a large datacenter network. However, since microbursts are unpredictable [32], identifying the specific packets involved in a microburst and exporting the corresponding queuing information requires going beyond the stateless "match and mirror" operation.

There have been systems proposed for monitoring a different class of microbursts, called *link utilization* microbursts [30, 32]. Link utilization microbursts are intervals where the utilization of a link exceeds a certain threshold, and unlike queuing microbursts, might not result in queuing. These systems [30, 32] can only detect *link utilization* microbursts at the time-scale of tens of microseconds. BurstRadar instead monitors *queuing* microbursts at a sub-microsecond resolution. It remains a future work to extend BurstRadar to monitor *link utilization* microbursts.

[2]Real-world microbursts queue lengths are less than 250 KB at the 90th percentile [32]. This requires Snappy to use smaller "window" sizes to better approximate the queue boundaries.

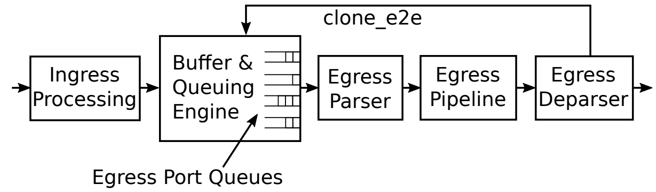


Figure 1: General architecture of a programmable switching ASIC [10]

3 BURSTRADAR

Our key idea is to first detect a microburst in the dataplane and then capture a *snapshot* of telemetry information of all the involved packets. This information allows queue composition analysis to identify the culprit flow(s), and burst profiling to know burst characteristics such as duration, queue build-up/drain rates, etc. Detecting a microburst is relatively easy with queuing telemetry information provided by modern programmable switching ASICs [25]. However, taking a *snapshot* of all the packets involved in the microburst and further exporting this information in an out-of-band manner is non-trivial for three reasons. First, the switching ASIC's "Buffer and Queuing Engine" (BQE) does not provide any support to peek into the contents of any queue, and so the *snapshot* needs to be captured from *outside* the BQE. Second, any logic in the programmable pipelines outside the BQE can only execute on a per-packet basis. Third, exporting the snapshot information requires on-demand generation of new *courier* packets in the dataplane and transferring of snapshot information to these *courier* packets.

Figure 1 shows the general architecture of a programmable switching ASIC. BurstRadar runs in the egress pipeline of each switch in the network. It consists of three functional components: (i) Snapshot Algorithm, (ii) Courier Packet Generation, and (iii) Ring Buffer. The Snapshot algorithm first determines the packets that are involved in a queuing microburst and *marks* them. The marking is done via a metadata header and does not modify the original packet. For each marked packet, a courier packet is generated to transport the marked packet's telemetry information via the switch's mirror port. The ring buffer provides temporary storage to facilitate the transfer of telemetry information from the *marked* packets to the courier packets. The telemetry information for each marked packet consists of the packet 5-tuple, the ingress and egress timestamps, and the queue depths at the time of enqueue and dequeue (`enqQdepth` and `deqQdepth`). Courier packets are processed at the monitoring servers connected to the mirror port infrastructure. In the following subsections, we describe these three components in more detail.

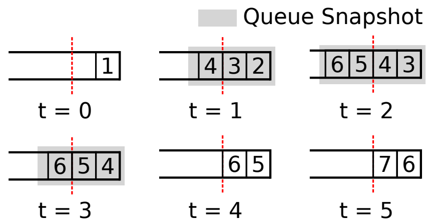


Figure 2: Evolution of an example queuing microburst at different instants in time

3.1 Snapshot Algorithm

While BurstRadar can monitor all queuing microburst events, small queuing events that cause negligible increase in application latency are generally not of interest to the operator. Therefore, BurstRadar allows the operator to specify a *latency-increase threshold* (specified as a percentage). For example, if the network's no-queuing RTT is 50 μs, then the operator may specify a threshold of 30% which translates to a minimum latency increase of 15 μs. BurstRadar would then ignore any microbursts that incur less than 15 μs of delay. The exact threshold to be set depends on the maximum delay that can be tolerated by the deployed applications. For latency-sensitive applications like web services, the threshold could be set to a small fraction of RTT. On the other hand, it could be a few multiples of RTT for throughput-intensive applications like Hadoop.

We define a *queue snapshot* to be the set of packets present in the queue when the queue-induced delay is above the operator-specified threshold. Figure 2 shows the evolution of a toy queuing microburst at different instants in time. At time instant t=1, the queue length exceeds the threshold (dotted line). Thus the snapshot of the queue at this instant consists of packets {2,3,4}. Similarly at t=2, the queue snapshot consists of packets {3,4,5,6}. At t=3, the queue starts to drain, but we still have a queue snapshot consisting of packets {4,5,6}. At t=4 and beyond, the queue length falls below the threshold and we stop taking snapshots. In other words, a single queuing microburst event consists of multiple overlapping queue snapshots. Note that at t=5, an additional packet #7 enters the queue but is not a part of any of the queue snapshots.

Since egress port queues are a part of the BQE (refer Figure 1), it would be easy to capture queue snapshots inside the BQE. However, the BQE in today's programmable switching ASICs doesn't provide any such functionality, but provides the queuing telemetry information (`enqQdepth` and `deqQdepth`) for each packet leaving the BQE. Our Snapshot algorithm uses this telemetry information and runs outside the BQE in the egress pipeline.

Since the egress pipeline follows a per-packet execution model, the Snapshot algorithm (Algorithm 1) needs to decide

Algorithm 1: Queue Snapshot Algorithm

```
  Input: threshold
  Initialization: bytesRemaining = 0;
1 foreach pkt in egressPipeline do
2     if deqQdepth > threshold then
3         bytesRemaining = deqQdepth − size(pkt);
4         mark(pkt);
5     else
6         if bytesRemaining > 0 then
7             bytesRemaining = bytesRemaining − size(pkt);
8             mark(pkt);
  end
```

(*mark*) whether a packet entering the egress pipeline belongs to any queue snapshot or not. To decide if a packet should be marked, we consider the queue length when a packet is dequeued (`deqQdepth`). There are two possible cases: (i) The `deqQdepth` is greater than the threshold, or (ii) The `deqQdepth` is less than or equal to the threshold. In the former, it is clear that the packet belongs to at least one of the queue snapshots. For example in Figure 2, packet #2's `deqQdepth` is greater than the threshold and thus packet #2 would be *marked*. In the latter case, our Snapshot algorithm tracks the `bytesRemaining` in the queue, each time a packet reports the `deqQdepth` to be greater than the threshold (line 3). When the reported `deqQdepth` is less than or equal to the threshold, only the packets equivalent of `bytesRemaining` would be marked (lines 6-8). In the example, packets #5 and #6 would be marked due to `bytesRemaining` set by packet #4; but packet #7 would not be marked.

3.2 Courier Packet Generation

BurstRadar generates a courier packet for each *marked* packet on demand. To do this, BurstRadar uses the *clone egress to egress* or `clone_e2e` primitive provided by programmable switching ASICs [10]. The `clone_e2e` primitive makes a copy of the exiting regular packet and places it in the egress queue of the mirror port (see Figure 1). The courier packet is also appropriately truncated to remove the original payload.

3.3 Ring Buffer

A ring buffer is designed using the transactional stateful memory available in the egress pipeline and exposed by the P4 programming language [3] as *register* arrays. The ring buffer acts as temporary storage for the telemetry information of marked packets until it can be copied into the courier packets.

Ring Buffer Sizing. We found that the first read from the ring buffer by a courier packet happens only after a *cloning delay*. During a microburst, the interval between the first and second writes to the ring buffer is mainly determined by

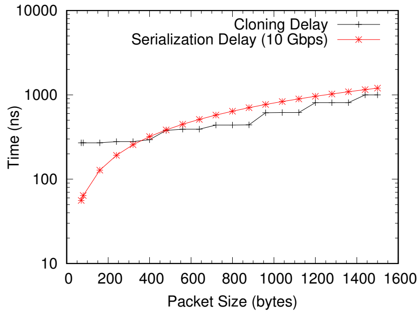


Figure 3: Cloning and serialization delay for packets of different sizes

the serialization delay of the first packet. If the serialization delay of the first packet is smaller than the cloning delay, the second packet in the microburst will write to the ring buffer before the first courier packet performs the first read. Therefore, the ring buffer needs to be large enough to store the information for the marked packets passing through the pipeline before the first read by the courier packets.

Figure 3 compares the cloning delay to the serialization delay (at 10 Gbps link speed) for packets of different sizes. For 64 byte packets, the cloning delay (270 ns) is more than five times the serialization delay (51.20 ns). This means that in the worst case of having all 64 byte packets in a microburst, more than five writes would be made to the ring buffer before the first read happens. For our ASIC implementation, we found that factors[3] other than the packet size also affect the cloning delay. Accordingly, we found (by measurement) the required minimum ring buffer sizes for 10 Gbps and 25 Gbps link speeds to be 26 entries and 32 entries, respectively. Since the size of each entry is 29 bytes, these requirements translate to 754 bytes and 928 bytes, which are very small given the SRAM memory sizes (up to 100 MB) in today's ASICs [22].

Concurrent Microbursts. Since the egress pipeline is shared among the ports, it serves the egress port queues in a round-robin manner. Therefore, if multiple egress ports simultaneously experience microbursts, in one scheduling round of the egress pipeline, there would be multiple writes to the ring buffer while only a single read due to a single mirror port. This seems to suggest that a very large ring buffer might be required to handle multiple concurrent microbursts. However, as we show in §4.2, in practice, a ring buffer with 1k entries is sufficient to handle 10 simultaneous microbursts without any overwrites.

3.4 Implementation

BurstRadar can be implemented with small modifications to fixed function switching ASICs or programmed on modern

[3]Investigation of all factors will be addressed in a separate measurement study.

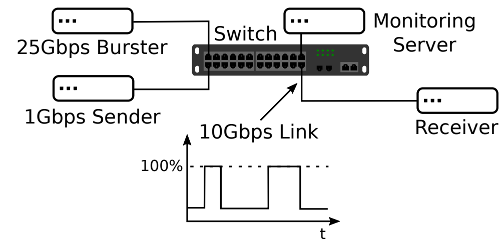


Figure 4: Testbed setup

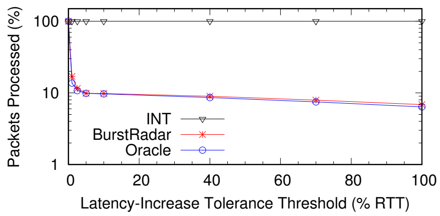
programmable switching ASICs [5, 15, 25]. We implemented BurstRadar on a Barefoot Tofino switch [25] in about 550 lines of P4 [3] code. The operator-specified latency-increase threshold is stored in a *register* in the dataplane and can be dynamically configured by the control plane. The Snapshot algorithm and the ring buffer are implemented using a sequence of exact match-action tables. Arithmetic operations are facilitated by stateful ALUs.

Switching ASICs (fixed or programmable) provision memory for buffered packets using fixed size memory buckets or *segments* [24]. Therefore, the reported `deqQdepth` is expressed in terms of number of segments. Snapshot algorithm converts the `deqQdepth` from segments to bytes to compute `bytesRemaining` (line 3 in Algorithm 1). This conversion results in excess `bytesRemaining` compared to the actual remaining bytes, causing BurstRadar to mark extra packets towards the *tail-end* (lines 6-8 in Algorithm 1) of a microburst. For example, if the segment size is 160 bytes and the queue consists of a single 161 byte packet, then the reported `deqQdepth` of 2 segments would be converted to 320 bytes, instead of the actual 161 bytes.

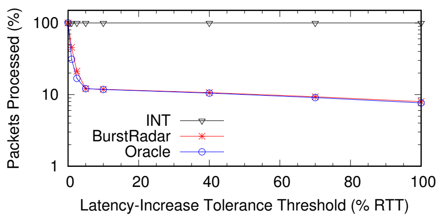
4 EVALUATION

The evaluation of our BurstRadar prototype is centered around answering three questions. First, how efficient is BurstRadar, given that it selectively *snapshots* microburst queues? Second, how well does BurstRadar handle multiple simultaneous microbursts? And finally, what is the cost of BurstRadar in terms of hardware resources required in the switching ASIC?

Testbed. The evaluation experiments were conducted in our hardware testbed which consists of a Barefoot Tofino [25] switch and commodity servers equipped with Intel XXV710 (25/10 Gbps) and Intel X710 (10 Gbps) NIC cards. To precisely generate network traffic at μs resolution and cause microbursts as per the input network traces, we wrote our own traffic generator application (450 lines of C++) using the PcapPlusPlus library [27] with DPDK [12] as the datapath. The testbed is organized in a topology as shown in Figure 4. Based on the data in [32], the sender continuously sends 1 Gbps background traffic keeping the utilization of the test link under 10% for 90% of the time. The burster emulates different sources of microburst, sending bursts at 25 Gbps



(a) Cache Traffic



(b) Web Traffic

Figure 5: Fraction of total number of packets processed for different latency-increase thresholds

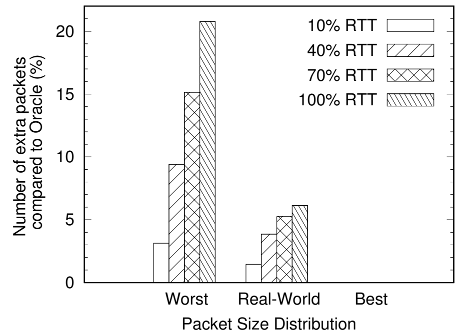


Figure 6: Number of extra packets marked compared to the Oracle solution for different packet size distributions (Cache Traffic)

such that the queuing at the switch's egress buffer (and the subsequent 100% link utilization) follows the distributions for duration and inter-arrival times as per the input trace.

Network Traces. Data on the frequency and duration of *queuing* microbursts is currently not available publicly. Therefore, we took reference from the data on *link utilization* bursts in a Facebook datacenter [32]. It provides the distribution of duration, inter-arrival times and packet size for utilization bursts when the link utilization spikes above 50%. We can safely assume that this is the worst case upper bound on the duration and inter-arrival times for *queuing* microbursts, which entail 100% link utilization on the egress link. We used the traffic data from two latency-sensitive applications – web, and in-memory cache – to generate 10-second long traces.

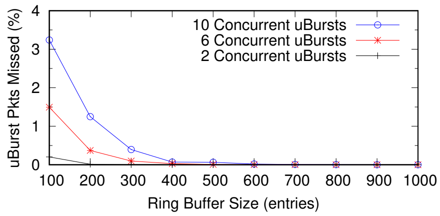
Methodology. We compare BurstRadar to INT [13, 20] and to an offline Oracle algorithm. The Oracle algorithm has access to the telemetry information of all the packets and is thus able to capture queue snapshots as if they were captured by the BQE (c.f. §3.1). It represents the optimal solution.

4.1 Efficiency

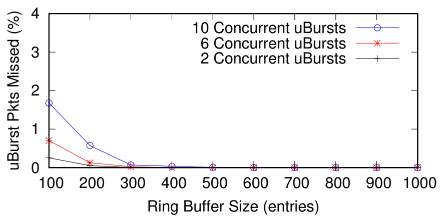
We quantify the overhead of continuous microbursts monitoring in terms of the fraction of total number of packets that are required to be processed by the monitoring system. We compare the overhead incurred by BurstRadar, INT and the Oracle algorithm for the cache and web traffic in Figure 5. We observe in Figure 5(a) that even for a low latency-increase threshold of 5% RTT, BurstRadar is 10 times more efficient than INT. Since the RTT is approximately 25 μs in our testbed, this threshold translates to 1.25 μs of queuing delay, or 1562.5 bytes worth of queuing at 10 Gbps. We verified with our experiments that this threshold only filters out packets that are not involved in microbursts. In practice, latency sensitive applications might not require such a low threshold and therefore the overhead for BurstRadar would be even lower. Note that at a latency-increase threshold of 0% RTT, BurstRadar would be equivalent to INT as telemetry information for every single packet is processed. The efficiency result for web traffic is similar to cache traffic as shown in Figure 5(b).

Overhead of Extra Packets. While BurstRadar is more efficient than INT in terms of the number of packets processed, it does process a few extra packets due to the segments to bytes conversion of `deqQdepth` (see §3.4). The number of extra packets identified by BurstRadar depends on the packet size distribution and the segment size of the ASIC's packet buffer. The worst case occurs when every packet is exactly one byte larger than the segment size, thereby causing each packet to occupy two segments. The best case occurs when the size of all packets is an integer multiple of the segment size.

In Figure 6, we plot the number of extra packets identified by BurstRadar compared to the Oracle algorithm for the cache workload with different packet size distributions: worst, real-world and best. For real-world, we use the cache workload's original packet size distribution [32]. We note that while the worst case shows about 21% extra packets compared to the Oracle, the number of extra packets is typically only about 6% as shown by the real-world case. In the best case, there are no extra packets. Figure 6 also shows that a larger latency-increase threshold leads to a higher proportion of extra packets. This is because for a given packet-size distribution, larger the latency-increase threshold, larger the number of packets in the *remaining* queue below the threshold, with each packet contributing extra bytes to `bytesRemaining`. The real-world overhead



(a) Cache Traffic



(b) Web Traffic

Figure 7: Fraction of microburst packets missed with concurrent microbursts for different ring buffer size

for web traffic is lower than the cache traffic due to larger packet sizes [32] and is omitted because of space constraints.

4.2 Handling Concurrent Microbursts

As discussed in §3.3, multiple concurrent microbursts can result in a higher rate of writes to the ring buffer than the rate of reads. If the ring buffer size is not sufficiently large, ring buffer overwrites may occur, leading to the loss of telemetry information for some of the *marked* packets. Currently, no data is available on how often we should expect concurrent microbursts at different egress ports for a switch. Therefore, we simulate[4] microburst traffic on multiple ports of a switch using 10-second long traces of cache and web traffic from [32]. In Figure 7, we plot the fraction of microburst packets missed by BurstRadar for different ring buffer sizes (10 to 1k entries) when microbursts occur on 2, 6 and 10 ports concurrently. With just 300 entries, BurstRadar is expected to be able to handle 10 simultaneous microbursts (cache traffic) with a packet miss rate lower than 1%. About 1000 entries are required to reduce the miss rate to absolute 0%. This suggests that BurstRadar is resilient and can handle simultaneous microbursts with a modestly-sized ring buffer.

4.3 Resource Utilization

In Table 1, we compare the hardware resources required by our BurstRadar prototype (with a ring buffer of 1k entries) to that required by a production (closed-source) version of switch.p4. The switch.p4 is a baseline P4 program that implements various networking features (L2/L3 forwarding, VLAN, QoS, ACL, etc.) for a typical datacenter ToR switch. A simplified open-source version of switch.p4 is available at [9]. We note that BurstRadar's overall resource consumption is low for various hardware resources. BurstRadar consumes a relatively larger proportion (12.5%) of stateful ALUs as they are used for the computations in our Snapshot algorithm and for managing the ring buffer pointers. The SRAM is used for the exact match-action tables and for implementing the ring buffer. Despite the ring buffer size of 1k entries, BurstRadar's SRAM requirements remain low. Also, the combined usage of all resources by switch.p4 and BurstRadar is well below 100%. This means that BurstRadar can easily fit on top of switch.p4.

[4]This experiment is currently not supported by our testbed due to lack of equipment to generate multiple concurrent microburst traffic.

Table 1: Hardware resource consumption of BurstRadar (ring buffer size of 1k entries) compared to the baseline switch.p4

Resource	switch.p4	BurstRadar	Combined
Match Crossbar	50.13%	3.39%	53.52%
Hash Bits	32.35%	4.83%	37.18%
SRAM	29.79%	4.06%	33.85%
TCAM	28.47%	0.69%	29.16%
VLIW Actions	34.64%	4.69%	39.33%
Stateful ALUs	15.63%	12.5%	28.13%

5 CONCLUSION

Detecting microbursts in a datacenter network and identifying the contributing flows is difficult because microbursts are unpredictable and last for 10's or 100's of μs. BurstRadar leverages programmable switching ASICs to implement continuous and efficient monitoring of microbursts by capturing the telemetry information of only the packets involved in microbursts. Our testbed evaluation using production network traces demonstrates that BurstRadar can detect microbursts with 10 times less overhead compared to existing approaches and is resilient to simultaneous microbursts. This paper describes our BurstRadar prototype, as well as the design decisions and considerations in dataplane packet cloning and ring buffer sizing. Our work demonstrates that modern programmable ASICs have made it practical to detect and characterize microbursts at multi-gigabit link speeds in high-speed datacenter networks. BurstRadar is a work in progress and we are working towards extending BurstRadar to detect link utilization microbursts.

ACKNOWLEDGEMENTS

This work was carried out at the SeSaMe Centre, supported by Singapore NRF under the IRC@SG Funding Initiative. It was also partly supported by the Singapore Ministry of Education tier 1 grant R-252-000-693-114. Ting Qu (201703170198) thanks the China Scholarship Council ([2017]3109) for financial support to visit the National University of Singapore.

REFERENCES

[1] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *Proceedings of SIGCOMM*.

[2] Theophilus Benson, Aditya Akella, and David A Maltz. 2010. Network traffic characteristics of data centers in the wild. In *Proceedings of IMC*.

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *SIGCOMM CCR* 44, 3 (2014), 87–95.

[4] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proceedings of SIGCOMM*.

[5] Cavium. 2018. XPliant Ethernet Switch Product Family. https://goo.gl/xzfLLo

[6] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, and Ori Rottenstreich. 2018. Catching the Microburst Culprits with Snappy. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*.

[7] Cisco. 2017. Monitor Microbursts on Cisco Nexus 5600 Platform and Cisco Nexus 6000 Series Switches. https://goo.gl/5Xxhpm

[8] Benoit Claise. 2004. Cisco Systems Netflow Services Export version 9. *RFC 3954* (2004). https://tools.ietf.org/html/rfc3954

[9] P4 Language Consortium. 2018. Baseline switch.p4. https://github.com/p4lang/switch

[10] P4 Language Consortium. 2018. Portable Switch Architecture. https://p4.org/p4-spec/docs/PSA.html

[11] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of NSDI*.

[12] The Linux Foundation. 2018. DPDK. http://dpdk.org/

[13] P4.org Applications Working Group. 2018. In-band Network Telemetry (INT) Dataplane Specification v1.0. https://goo.gl/HtPE9K

[14] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. 2014. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks.. In *Proceedings of NSDI*.

[15] Intel. 2018. FlexPipe. https://goo.gl/PzPudG

[16] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. 2014. Millions of little minions: Using packets for low latency network programming and visibility. In *Proceedings of SIGCOMM*.

[17] Juniper Networks. 2016. Network Analytics Overview. https://goo.gl/TbNwSC

[18] Zaid Ali Kahn. 2016. Project Falco: Decoupling Switching Hardware and Software. https://goo.gl/U7PUQZ

[19] Rishi Kapoor, Alex C Snoeren, Geoffrey M Voelker, and George Porter. 2013. Bullet trains: a study of NIC burst behavior at microsecond timescales. In *Proceedings of CoNext*.

[20] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. 2015. In-band network telemetry via programmable dataplanes. In *Proceedings of SIGCOMM (Poster)*.

[21] Richard Martin. 2007. Wall Street's Quest To Process Data At The Speed Of Light. Information Week.

[22] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of SIGCOMM*.

[23] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-directed hardware design for network performance monitoring. In *Proceedings of SIGCOMM*.

[24] Arista Networks. 2015. Latency Analyzer (LANZ) Architectures and Configuration. https://goo.gl/LrRNi4

[25] Barefoot Networks. 2018. Tofino. https://goo.gl/cdEK1E

[26] Peter Phaal. 2004. sFlow. http://sflow.org/sflow{_}version{_}5.txt

[27] Seladb. 2018. PcapPlusPlus. https://github.com/seladb/PcapPlusPlus

[28] Danfeng Shan, Fengyuan Ren, Peng Cheng, and Ran Shu. 2016. Microburst in Data Centers: Observations, Implications, and Applications. *arXiv preprint arXiv:1604.07621* (2016).

[29] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. 2015. Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network. In *Proceedings of SIGCOMM*.

[30] Frank Uyeda, Luca Foschini, Fred Baker, Subhash Suri, and George Varghese. 2011. Efficiently Measuring Bandwidth at All Time Scales. In *Proceedings of NSDI*.

[31] Amin Vahdat. 2017. ONS Keynote: Cloud Native Networking. https://youtu.be/1xBZ5DGZZmQ

[32] Qiao Zhang, Vincent Liu, and Hongyi Zeng. 2017. High-Resolution Measurement of Data Center Microbursts. In *Proceedings of IMC*.

[33] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. 2015. Packet-level telemetry in large datacenter networks. In *Proceedings of SIGCOMM*.